RL-TR-95-22
Final Technical Report
March 1995

PLAN DEBUGGING USING APPROXIMATE DOMAIN THEORIES

Stanford University

Sponsored by
Advanced Research Projects Agency and Rome Laboratory
ARPA Order No. 7727

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-22 has been reviewed and is approved for publication.

APPROVED: Northrup Fowler

NORTHRUP FOWLER III, Ph.D.
Project Engineer

FOR THE COMMANDER: Henry J. Bush

HENRY J. BUSH
Acting Deputy Director
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3C) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

PLAN DEBUGGING USING APPROXIMATE DOMAIN THEORIES

Matthew L. Ginsberg

Contractor: Stanford University
Contract Number: F30602-91-C-0036
Effective Date of Contract: 8 February 1991
Contract Expiration Date: 7 November 1994
Short Title of Work: Plan Debugging Using Approximate Domain Theories
Period of Work Covered: Feb 91 - Nov 94

Principal Investigator: Matthew L. Ginsberg
Phone: (503) 346-0471
RL Project Engineer: Northrup Fowler III, Ph.D.
Phone: (315) 330-3011

This research was jointly sponsored by the Advanced Research Projects Agency of the Department of Defense and by Rome Laboratory, and was monitored by Northrup Fowler III, RL (C3C), 525 Brooks Rd, Griffiss AFB NY 13441-4505.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
March 1995

3. REPORT TYPE AND DATES COVERED
Final   Feb 91 - Nov 94

4. TITLE AND SUBTITLE

PLAN DEBUGGING USING APPROXIMATE DOMAIN THEORIES

5. FUNDING NUMBERS
C - F30602-91-C-0036
PE - 62301E & 62702F
PR - G727
TA - 00
WU - 05

6. AUTHOR(S)

Matthew L. Ginsberg

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Stanford University*
Stanford CD 94305-2140

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Rome Laboratory (C3C)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-95-22

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Northrup Fowler III, Ph.D./C3C/(315) 330-3011
*Stanford University is the prime contractor. The report was written at the University of Oregon, Eugene OR 97403, a subcontractor on this effort.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The overall goal of this project was to investigate the idea that planning in the presence of uncertain information could be easier, not harder, than planning in its absence. The basic reason proposed was that, since it is impossible to control planning search with precision, uncertain domain information had a potential for use in this area that more accurate information might lack. This suggestion turned out to be correct. During the course of the project, we developed compelling theoretical arguments showing that reasoning can be controlled more generally by using uncertain domain information than by using existing techniques such as hierarchy. Doing so, however, involves surmounting substantial practical difficulties in terms of both the underlying planning technology and our ability to manipulate uncertain information itself. This report describes these theoretical arguments and the problems (and potential solutions) that an implementation exploiting our ideas must face.

14. SUBJECT TERMS
Real-time reasoning, Plan debugging, Planning with uncertainty, Approximate planning

15. NUMBER OF PAGES
162

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Plan Debugging Using Approximate Domain Theories

Final Report

Contents

Abstract

The overall goal of this project was to investigate the idea that planning in the presence of uncertain information could be *easier*, not harder, than planning in its absence. The basic reason proposed was that since it is impossible to control planning search with precision, uncertain domain information had a potential for use in this area that more accurate information might lack.

This suggestion turned out to be correct. During the course of the project, we developed compelling theoretical arguments showing that reasoning can be controlled more generally by using uncertain domain information than by using existing techniques such as hierarchy. Doing so, however, involves surmounting substantial practical difficulties in terms of both the underlying planning technology and our ability to manipulate uncertain information itself. This report describes these theoretical arguments and the problems (and potential solutions) that an implementation exploiting our ideas must face.

1 Introduction

Our goal in this project was to investigate a simple idea: The uncertain information present in many planning domains can be used to focus search in these domains. The emphasis of our work was on default information: Is it possible to build a planner that uses this information to focus its search and thereby reduces the time needed to solve a particular planning problem?

The answer is a qualified yes. Yes because we were successful in our attempts to build such a planner; qualified because the planner itself is very unlike existing AI planners such as SIPE [106] or O-PLAN [9]. This report describes both the success and the qualification. The success is epistemological, while the qualification is heuristic.

From an epistemological point of view, our argument that defaults should be used to control planning search will proceed in three stages:

1. We will show that all control reasoning is fundamentally an appeal to base-level information about the structure of our knowledge in a particular domain. The control information used in SIPE, for example, is essentially a stand-in for domain information that could be described directly.

2. We will provide a mechanism whereby defaults can be viewed as shorthand for this sort of structural information.

3. We will show that the default description is substantially more powerful than its conventional counterpart. More specifically, we will show that a default description of a planning hierarchy has a flexibility lacked by the static hierarchies used by existing planning systems.

This argument is expanded and made precise in the next section of this report. Section 2.1 describes control reasoning as an appeal to base-level information, Section 2.2 explains how defaults specifically can be used in this context, and Section 2.3 shows that the default approach is properly more powerful than existing ones.

These epistemological arguments provide a compelling suggestion that generative planning systems solving large problems will need to exploit the control information implicit in uncertain domain information. Unfortunately, there are significant problems in doing so. These involve both the structure of modern generative planning systems and the difficulty of reasoning with default information generally.

With regard to planning, a planner exploiting default information for control purposes needs to satisfy three related criteria:

1. *It needs to plan for subgoals separately.* Because defaults lead to a flexible planning hierarchy, the control information implicit in uncertain information may well lead the planner to expand one portion of a plan at one point, and a separate portion of the plan subsequently. From an intuitive point of view, this sort of approach is essential; planning *should* be a dynamic process where the computational focus shifts from one area to another as domain information and the current state of the analysis warrant.

   If this approach is to be computationally viable, however, it must be *possible* to analyze first one portion of a plan and then another; existing generative planners are committed to static hierarchies that involve gradual expansion and analysis of single sections of the plan.

Planning for subgoals separately has other computational benefits that are recognized throughout the planning community (as we will discuss in Section 3.3); our approach is simply the first where this desirable feature has become essential.

2. *It needs to be declarative.* Existing generative planners have some declarative features, but are fundamentally state-based search engines exploring a space of possible plans. Control information is expressed in terms of the portion of the search space that should be examined at any particular juncture.

   Default information used for control purposes is not like this; it is declarative information that interacts at a fundamental level with the information in the planning problem itself. If the control of search is to exploit declarative information about the domain, the domain itself must clearly be declarative in nature.

   Once again, the feature that we are proposing is already of recognized value. Planners need to be declarative because things go wrong; it is inevitable that a planner embedded in a complex environment will need to stop and "figure out what is going on." This sort of figuring out is possible in a declarative setting but not in others.

3. *It needs to be interruptible.* This almost goes without saying: To control a computational process involves interrupting it. Here, too, we are advocating a property of recognized worth; many realistic planning applications involve real-time environments that require interruptibility.

Implemented solutions to the above difficulties are the topic of Section 3 of this report. We begin in Section 3.1 by presenting a general framework in which any declarative system can be made interruptible. Although not specific to planning, the technical ideas presented in this section underpin the work that follows. In Section 3.2, we go on to describe computational issues that must be addressed if a planner is to reason declaratively. In Section 3.3, we use these ideas and others to develop an interruptible, declarative planner that analyzes subgoals separately.

The other heuristic difficulties the project faced involved problems of reasoning with defaults generally. It is fairly well recognized that default reasoning is closely related to truth maintenance techniques [32]. In the predicate case, for example, they occupy identical locations in the polynomial hierarchy. Implementations of nonmonotonic reasoning systems typically appeal to truth maintenance techniques in one way or another [34].

Unfortunately, truth maintenance techniques have not been applicable in backward-reasoning systems because they typically require an amount of memory linear in the run time of the system in question. A secondary thrust of our research was therefore addressed at overcoming this difficulty. We developed a novel approach called *dynamic backtracking* that appears likely to enable the effective use of justification information in controlling search in declarative domains. Dynamic backtracking is described in Section 4; the crucial insight is that not all of the justifications developed by a backward reasoning system need to be retained in order for substantial theoretical and computational benefits to be realized. It promises to have significant impact in the constraint-satisfaction subfield of artificial intelligence, and represents ideas to which we were led by our work on plan control using default information. The ideas have not been incorporated into our generative planner, however, and the presentation given here reflects that. Dynamic backtracking generally is currently under further development at CIRL and elsewhere.

The broad outline of this report has already been described. Epistemological arguments are made in Section 2, while heuristic concerns are the focus of Sections 3 (planning) and 4 (truth maintenance). Concluding remarks are contained in Section 5.

One of the attractive properties of the reseach we have conducted is the fact that it will apply to a range of planning and scheduling problems wider than that found in transportation scheduling alone. Because of this, we were torn when writing this report between a desire to present examples showing the ideas in their full generality and a need to make clear their relevance to the transportation planning initiative specifically. We chose to present the scientific concepts generally, and the ideas in the body of the paper are therefore illuminated with examples drawn more frequently from commonsense domains than from transportation problems. Subsection 5.2 of the conclusion, however, corrects this by discussing the specific transportation planning impact of our principal scientific contributions.

2 Epistemological results

2.1 Control reasoning as an appeal to base-level information

The split between *base-level* and *metalevel* knowledge has long been recognized by the declarative community. Roughly speaking, base-level knowledge has to do with information about some particular domain, while metalevel knowledge has to do with knowledge *about* that information. A typical base-level fact might be, "Iraq invaded Kuwait," while a typical metalevel fact might be, "To show that a country c is aggressive, first try to find another country that has been invaded by c."

Base-level information is of necessity domain-dependent, since the facts presented will involve the particular domain about which the system is expected to reason. Metalevel information, however, can be either domain-dependent (as in the example of the previous paragraph and as typically described elsewhere [91]), or domain-independent. Typical domain-independent metalevel rules are the cheapest-first heuristic or the results found in Smith's work on control of inference [94, 95, 96].

In this section, we make an observation and a claim. The observation is that there are in fact two distinct types of metalevel information. On the one hand, one can have metalevel information about one's base-level knowledge itself; on the other, one can have control information about what to *do* with that knowledge.

This is a distinction that has typically been blurred by the AI community; we intend to focus on it here. We will refer to the second type of metalevel knowledge as *control* knowledge, and will refer to knowledge about knowledge (and not about what to do with that knowledge) as *modal*. We choose this term because sentences expressing modal knowledge typically involve the use of predicate symbols that have other sentences as arguments. Thus a typical modal sentence might be, "I know that Iraq invaded Kuwait," or "I don't know of anyone that Kuwait has invaded." Note that the information here doesn't refer to the domain so much as it does to our knowledge *about* the domain; nor is it control information telling us what to do with this knowledge. It simply reports on the state of our information at some particular point in time. In general, we will describe as modal all information describing the structure of our declarative knowledge in some way.

We will show that there is no fundamental place in declarative systems for domain-dependent control information. Rather, we suggest that every bit of information of this sort is in fact a conflation of two separate facts – a domain-independent control rule and a domain-dependent modal fact telling us that the rule can be applied. A similar observation has already been made by David Smith:

> Good control decisions are not arbitrary; there is always a reason why they work. Once these reasons are uncovered and recorded, specific control decisions will follow logically from the domain-independent rationale, and simple facts about the domain. [93, p. 12]

This is an observation with far-reaching consequences, including the following:

1. There is no need for a "metametalevel" or anything along those lines. Domain-independent control information need not be refined using higher-order information. A similar conclusion has been reached in a decision-theoretic setting by Lipman [64].

2. Current work on learning need not focus on the development of new control rules [68], but should instead focus on the development of new base-level or modal information. This may make the connection to existing ideas (such as caching) somewhat easier to exploit and understand.

3. Current work on the control of inference should restrict its focus to domain-independent techniques. There is much to be done here; Smith's work only scratches the surface. At a minimum, general-purpose information can be extracted from the domain-specific control rules currently incorporated into many declarative systems.

4. The recognition that the control of reasoning should proceed by applying domain-independent rules to domain-dependent modal and base-level information can enhance the power of systems that control reasoning in this way. This is a consequence of the fact that many declarative databases already contain such domain-dependent information without exploiting its implications for search control. In Section 2.2, we suggest that the real role of nonmonotonic reasoning in AI is to focus inference, and that suggestion is typical of this proposal.

The outline of this section of the report is as follows: We begin in Section 2.1.1 with an example, showing the replacement of a specific control rule by a domain-independent one using domain-dependent modal information. We then go on to show that this procedure generalizes with minimal computational cost to all domain-specific control information.

The general construction is extremely weak, and Section 2.1.2 examines a more powerful application related to work described subsequently and elsewhere [20, 68, and Section 2.2]. Remarks concerning the future of these ideas are contained in Section 2.1.3.

2.1.1 The basic result

Consider the following declarative database, written in PROLOG form:

```
hostile(c) :- allied(d,c), hostile(d).
hostile(c) :- invades-neighbor(c).
allied(d,c) :- allied(c,d).
invades-neighbor(Iraq).
allied(Iraq,Jordan).
```

Informally, what we are saying here is that a country is hostile if it is allied to a hostile country, or if it invades its neighbor. Alliance is commutative, and Iraq has invaded its neighbor and is allied with Jordan.

Now suppose that we are given the query hostile(Jordan); is Jordan hostile? In attempting to prove that it is, it is important that we eventually expand the subgoal invades-neighbor(Iraq) rather than only pursue the infinite chain of subgoals of the form

```
hostile(Jordan),hostile(Iraq),hostile(Jordan),hostile(Iraq),...
```

We might describe this in a domain-dependent fashion by saying that subgoals involving invades-neighbor should be investigated before subgoals involving hostile. (There are obviously many other descriptions that would have the same effect, but let us pursue this one.)

Now consider the following geometric example:

```
acute(t) :- congruent(u,t), acute(u).
acute(t) :- equilateral(t).
congruent(u,t) :- congruent(t,u).
equilateral(T1).
congruent(T1,T2).
```

A triangle is acute if it is congruent to an acute triangle or if it is equilateral. Triangles T_1 and T_2 are congruent, and T_1 is equilateral. Is T_2 acute?

This example is different from the previous one only in that the names of the predicate and object constants have been changed; from a machine's point of view, this is no difference at all. It follows from this that if the control rule delaying subgoals involving the predicate hostile is valid in the initial example, a rule delaying subgoals involving acute will be valid in the new one.

What we see from this is that the control rules are not operating on base-level information so much as they are operating on the *form* of our declarative database. To make this remark more precise, we need to discuss the structure of the proof space a bit more clearly. We will do this by axiomatizing a description of this proof space. This axiomatization will refer to the structure of the proof space only and will therefore be modal information in the sense described earlier; since nothing will be said about how to search the proof space, no domain-dependent control information will be used.

The language we will use will label a node in the proof space by a set $\{p_1, \ldots, p_n\}$, where all of the p_i need to be proved in order for the proof to be complete. A node labelled with the empty set is a goal node.

We will also introduce a predicate child, where $\texttt{child}(n_1, n_2)$ means that the node n_2 is a child of the node n_1. Thus in the original example, we would have

$$\texttt{child}(\{\texttt{hostile}(c)\}, \{\texttt{invades-neighbor}(c)\}) \tag{1}$$

saying that a child of the node trying to prove that c is hostile is a node trying to prove the subgoal that c has invaded its neighbor.

We can go further. By taking into consideration all of the facts in the database together with the inference method being used, we can delimit the extent of the child predicate exactly. This eventually leads to a large disjunction, one term of which is given by (1):

$$\begin{aligned} &\texttt{child}(m, n) \equiv \\ &\quad [m = n \cup \{\texttt{invades-neighbor(Iraq)}\}] \vee \\ &\quad [m = n \cup \{\texttt{allied(Iraq, Jordan)}\}] \vee \\ &\quad \exists S.[m = S \cup \{\texttt{allied}(x, y)\} \wedge n = S \cup \{\texttt{allied}(y, x)\}] \vee \ldots \end{aligned} \tag{2}$$

The above expression refers to the predicate and object constants appearing in the initial database, in this case hostile, invades-neighbor, allied, Iraq and Jordan. We can abbreviate the large expression appearing in (2) to simply

$$\texttt{type}_{37}(\texttt{hostile}, \texttt{invades-neighbor}, \texttt{allied}, \texttt{Iraq}, \texttt{Jordan}) \tag{3}$$

where we are using the subscript to distinguish databases of this form from databases of some other form. Note that (3) is in fact a consequence of the form of the information in our database, so that we can either derive (3) before applying a (domain-independent) control rule in which $\texttt{type}_{37}$ appears, or derive and stash (3) when the database is constructed. The second example is similar, allowing us to derive

$$\texttt{type}_{37}(\texttt{acute}, \texttt{equilateral}, \texttt{congruent}, T_1, T_2)$$

At this point, the hard work is done – we have formalized, via the type predicate, our observation that the two databases are the same. The control rule that we are using is now simply

$$\texttt{type}_{37}(p_1,p_2,p_3,o_1,o_2) \supset \texttt{delay}(p_1) \tag{4}$$

indicating that for databases of this type, with the p_i being arbitrary predicate constants and the o_i being arbitrary object constants, subgoals involving the predicate p_1 should be delayed. Of course, we still need to interpret delay in a way that will enable our theorem prover to make use of (4), but this is not the point. The point is that (4) is no longer a domain-specific control rule, but is now a domain-independent one. Any particular application of this domain-independent rule will make use of the modal information that a given database satisfies the predicate $\texttt{type}_{37}$ for some particular choice of constants. As we have already noted, this modal information can either be derived as the theorem prover proceeds or can be cached when the database is constructed.[1]

Proposition 2.1 *Any domain-dependent control rule can be replaced with a domain-independent control rule and a modal sentence describing the structure of the search space being expanded by the theorem prover. The computational cost incurred by this replacement is that of a single inference step, and the domain-independent control rule will be valid provided that the domain-dependent rule was.*[2]

2.1.2 A more interesting example

Although correct, the construction in the previous section is in many ways trivial, since in order to transfer a control rule from one domain to another we need to know that the two search spaces are identical. Indeed, this is the only circumstance under which we can be assured that the rule will remain valid when the domain changes.

The reason that our earlier construction is interesting, however, is that the intuitive arguments underlying domain-dependent control rules do *not* typically depend on the complete structure of the search space. In this section, we will consider a more typical example due to David Smith.

The example involves planning a trip to a distant location; let us suppose from Stanford to MIT.[3] The domain-dependent rule Smith presents is the following: When planning a long trip, plan the airplane component first. Why is this?

There are two reasons. Suppose that we form the tentative high-level plan of driving from Stanford to San Francisco airport, flying from there to Logan, and then driving to MIT. The decision to plan the airplane component of the trip first is based on the observations that:

[1] For efficiency reasons, we might want to use the control rule only when the database is *known* to be of this type. This could be encoded by using a modal operator of explicit knowledge to replace (4) with a more suitable expression.

[2] This report does not contain proofs of the results stated; the material we are presenting has appeared or will appear elsewhere and the interested reader is referred to these other publications if proofs are needed. Results in this section are proved in [45].

[3] Why anyone would actually want to *make* this trip escapes us.

1. The three legs of the journey are probably noninteracting. Except for scheduling concerns, the nature of our transportation to and from the airport is unrelated to our flight from San Francisco to Boston. It therefore makes sense to plan for each of the subgoals separately.

2. Airplane flights are tightly scheduled, whereas ground transportation is typically either loosely scheduled (because busses run frequently) or not scheduled at all (if we propose to drive to and from the airports involved). If we schedule the ground transportation first, we may be unable to find a flight that satisfies the induced constraints.

The observation that our three subgoals (drive to SFO, fly to Logan, drive to MIT) do not interact is little more than an application of the frame axiom, which says that once a subgoal has been achieved it will remain satisfied. (So that renting a car from Hertz will not magically teleport us from Boston to Miami; nor will driving to the airport cause United to go bankrupt.) What we are doing here is applying the control rule:

> When attempting to achieve a conjunctive goal, it is reasonable to attempt to achieve the conjuncts separately.

Elkan also makes this observation [20].

It is a default base-level fact – the frame axiom – that justifies our confidence in this domain-independent control rule.[4] The frame axiom justifies our belief that we will be able to achieve the subgoals separately, and therefore that planning for them separately is a computationally useful strategy. Elkan's principle is in fact a special case of the approach that we are proposing, an observation we will extend shortly by suggesting that the true role of nonmonotonic reasoning in AI generally is to enable us to simplify problems in this way.

The reason that we plan for the airplane flight before planning for the other two subgoals is similar. Here, we note that solving the subgoal involving the airplane flight is unlikely to produce a new problem for which no solution exists, while solving the subgoals involving surface transportation may. So we are invoking the domain-independent principle:

> When solving a problem, prefer inference steps that are unlikely to produce insoluble subproblems.

This domain-independent control information is applied to the domain *dependent* modal information that

$$\texttt{fly}(\texttt{SFO}, \texttt{Logan}, t)$$

is unlikely to have a solution if t is bound, while

$$\texttt{drive}(\texttt{Stanford}, \texttt{SFO}, t)$$

[4] In fact, this control rule is not completely domain-independent, since it applies to planning only. But it can easily be extended to the completely domain-independent rule that when attempting to solve a problem, if there is good reason to believe that the solution to a simpler problem will suffice, one should solve the simpler problem and then check to see if it is a solution to the original query.

is likely to have a solution whether t is bound or not. Once again, we see that we are applying a domain-independent control rule to domain-dependent modal knowledge. Furthermore, the computational arguments in Proposition 2.1 remain relevant; it is no harder to cache partial information regarding the structure of the proof space (e.g., $\texttt{fly(SFO}, \texttt{Logan}, t)$ is unlikely to have a solution if t is bound) than it is to cache a complete description as in (3).

An example due to Minton [68] can be handled similarly. This example is from the blocks world, where Minton suggests that when planning to build a tower, expect to build it from the bottom up. We have chosen to discuss Smith's example in detail because only his rule is pure control information.[5] In Minton's case, the fact that you will *build* the tower from the bottom up is no reason to do the *planning* by beginning with consideration of the bottom block, although Minton appears to use the rule in this fashion. After all, it is easy to imagine domains in which the nature of the tower is constrained by the properties of the uppermost object it contains; in this case it might well be advisable to do the planning from the top down even though the *construction* will inevitably be from the bottom up. Smith's example does not share this ambiguity; it really is the case that one plans long trips by beginning with any airplane component.

We should be careful at this point to be clear about what distinguishes the examples of this section from those that appear in Section 2.1.1. The basic result of Section 2.1.1 was the following:

> By constructing control rules that appeal to the precise form of a declarative database and by caching modal information about the form of the database being used in any particular problem, control rules can always be split into domain-independent control knowledge and domain-dependent base-level or modal knowledge. Further, this split incurs minimal computational expense.

The upshot of the examples we have discussed in this section is the following:

> Although the control rules used by declarative systems might in principle be so specific as to apply to only a single domain, the domain-independent control rules in *practice* appear to be general enough to be useful across a wide variety of problems.

Whether or not this observation is valid is more an experimental question than a theoretical one; if the observation *is* valid, it reflects that fact that our world is benign enough to allow us to develop control rules of broad merit. The evidence that we have seen indicates that our world is this benign; all of the examples of control information that we have been able to identify in the AI literature (or in commonsense discourse) appeal to domain-independent control principles that have wide ranges of validity. Nowhere do we see the use of a control rule whose justification is so obscure that it is inaccessible to the system using it.

Other results [23] also lend credence to this belief. Etzioni's STATIC system showed that determining whether or not a rule learned by PRODIGY was likely to be useful in

[5] As Smith himself has pointed out (personal communication).

practice could be determined by a static examination of the PRODIGY search space. Since Etzioni's system examines PRODIGY's search space only to determine whether or not it is recursive, we see once again a general domain-independent rule (involving the need to avoid recursive problem descriptions) being used to construct apparently domain-specific control information.

2.1.3 Impact

Our aim in this section of the report has been to argue that reasoning systems have no need for domain-dependent control rules; rather, they apply domain-independent control techniques to domain-dependent modal and base-level information. In retrospect, the viability of this approach is obvious; what is interesting are the consequences of this view.

Higher orders of meta One of the difficulties with control reasoning thus far has been that it seems to require a "metametalevel" in order to decide how to reason about the control information itself, and so on.

Domain-independent control information eliminates this philosophical difficulty. Since the control information is completely general, there is no reason to modify it in any way. The implemented system must be faithful to this control information but need have no other constraints placed on it.

Another way to look at this is the following: Provided that no new control information information is learned, existing metalevel information can be "compiled" in a way that eliminates the need for runtime evaluation of the tradeoffs between competing control rules. The observations we have made allow us to conclude that the new control information learned by existing systems is in reality base-level or modal information instead; viewing it in this way allows us to dispense with any reexamination of our existing control decisions. This reexamination is, of course, the "metametalevel" analysis that we are trying to avoid.

Research on control of inference There has been little work to date on domain-independent techniques for the control of search. Smith investigates the question in a declarative setting [94, 95, 96], while authors concerned with constraint satisfaction have considered these issues from a somewhat different viewpoint [13, 14, 15, 44, and others].

Our ideas suggest an obvious way in which these results can be extended. If the existing domain-dependent control information used by various systems does in fact rest on general domain-independent principles, these general principles should be extracted and made explicit. In many cases, doing so may involve extending the declarative language being used to include information that is probabilistic [94, 95, 96] or default [20, and Section 2.2] in nature; this can be expected to lead to still further interesting research questions.

Domain-dependent information The recognition that the domain-dependent information used by a declarative system is all base-level or modal has interesting implications in its own right.

The most immediate of these is the recognition that learning systems should not be attempting to learn control rules directly, but should instead be focusing on the base-level information that underlies them. Thus a travel-planning system should be trying to recognize that airline flights are scheduled while automobile trips are not, as opposed to the specific rule that one should plan around airline flights.

There are some immediate benefits from this approach. The first is that domain-independent control rules may be able to make use of domain-dependent information that has been included in the system for reasons having nothing to do with the control of search. In Section 2.1.2, for example, we saw that the frame axiom, through its implication that conjunctive subgoals can be achieved separately, had significant consequences in planning search. These consequences are obtained automatically in our approach.

A second benefit is that learned modal knowledge can be used by any domain-independent control rules included in the system. If domain-specific control rules are learned instead, other consequences of the modal knowledge underlying these rules will not be exploited.

Third, subtle interactions among domain-dependent control rules can be understood in terms of simpler base-level information. Thus if my reason for going to MIT is to give a presentation of some sort, I will typically plan the time of the presentation first, only then scheduling the airplane and other parts of my travel. The reason is the base-level fact that the folks at MIT are likely to be even more tightly scheduled than are the airlines. The apparent conflict between the control rules, "Plan airplane flights first," and "Plan talks first," is recognized as tension between the base-level facts, "Airplane flights are tightly scheduled," and "Academic talks are tightly scheduled." Assuming that our declarative language is able to compare the truth values assigned to these two latter sentences, the conflict will be resolved in a straightforward fashion.

Finally, a focus on the ability to derive base-level information may allow us to address an aspect of human problem solving that has been ignored by the AI community thus far. Specifically, the results of the problem-solving effort themselves frequently impact our base-level expectations. As an example, after an hour spent fruitlessly searching for a proof of some theorem, my belief in the theorem may waver and I may decide to look for a counterexample. What I am doing here is changing my base-level beliefs based not on new information, but on the results of my theorem-proving efforts directly; the new base-level expectations then refocus my attention via their relationship to domain-independent rules (here, information about when one should try to prove a theorem and when one should attempt to prove its negation). Domain-independent control information that is capable of dealing with our new base-level expectations will duplicate behavior of this sort.[6]

In all of these cases, the advantages we have identified are an immediate consequence of the observation that reasoning systems should work with a domain *dependent* base level (including modal information) and domain *independent* control information.

[6] The blind search technique of iterative broadening [46] is based in part on a similar observation.

2.2 Control of reasoning using defaults

In the applications that have been proposed for nonmonotonic reasoning, it is possible to draw default conclusions quickly; the computational expense of the default approach comes from the need for some sort of consistency check. In most cases, it is necessary to check that the default conclusions are consistent with the initial database.

A default approach to reasoning about action, for example, requires two invocations of a theorem prover in order to determine whether or not some domain fact is preserved when an action is executed. In the first, we simply use the frame axiom to conclude that it *is* preserved; in the second, we check the consistency of this conclusion.

Here, the first "proof" consists simply of an application of the frame axiom; the work is in the consistency check. Since a monotonic approach to reasoning about action involves checking a variety of persistence conditions in order to determine that the domain fact persists through the action, it is possible for the nonmonotonic approach to be the more efficient of the two. This argument is made quantitative elsewhere [49].

Even though the nonmonotonic approach requires an additional call to a theorem prover, it is possible for it to be more efficient than the conventional one because each of the two proof attempts may be much more efficient than its monotonic counterpart. In this section, we suggest that this is the true role of nonmonotonic reasoning in artificial intelligence: the replacement of one complex proof with two or more simpler ones.

The computational benefits of this sort of approach are well known. Consider the basic idea underlying island-driven search [21] or hierarchical planning [83]. By identifying an island through which a plan or search must pass, it is possible to split one search problem into two smaller ones; since the difficulty of the problem grows exponentially with its size, this leads to computational savings. We expand on this particular application in Section 2.3.

In general, it is possible that we are not *certain* that the solution to a given search problem passes through some specific island, but only *expect* it. In this case, we search first for a solution passing through the proposed island, and subsequently for one that avoids it. Provided that the cost of identifying the hypothetical island is outweighed by the expected gain in solving the smaller search problem, the identification of the island is a justifiable expense.

As we will see, the identification of the island corresponds to the discovery of a default solution to the problem, and the subsequent consistency check has an analog in the search for a solution that passes through the island in question. This allows us to rephrase the observation in the previous paragraph as follows: Provided that the cost of the first default proof is outweighed by the expected gain in checking the consistency of this default proof as opposed to solving the original problem, the use of default information is a justifiable expense.

In the next section, we will discuss this issue in terms of an extended example involving the use of an ATMS to solve a synthesis problem. A discussion of the computational advantages offered by this idea is contained in Section 2.2.2. Planning examples are discussed in Section 2.2.3, and an examination of the nature of the "consistency check" required by existing nonmonotonic formalisms is the topic of Section 2.2.4.

2.2.1 Synthesis problems

Our focus here will be on synthesis problems, such as planning or circuit design. One possible implementation of a planning system, for example, uses resolution to prove that there exists a situation in which the goal has been achieved and then recovers the actions needed to achieve the goal by examining the situation so constructed [50].

Another possible approach [26] takes as input some (probably inconsistent) set A of assumptions, and looks for a consistent subset $S \subseteq A$ that allows us to conclude that the goal has been achieved. The assumption set A might consist of a variety of assumptions about which actions were taken at what times; if A includes both the assumption that we crossed the street at time t and the assumption that we remained stationary at that time, A itself will be inconsistent and we will need to commit to at most one of these actions when actually constructing our plan.

Of course, finding a consistent subset of A that entails some specific goal g is what ATMS's are all about [81]. (The restriction that S above be consistent is the same as de Kleer's condition that S not contain a nogood [10].) We will refer to this type of problem in general as follows:

Definition 2.2 *Suppose that we have fixed a base theory T, a set of assumptions A and a goal g. We will call the problem of finding a subset $S \subseteq A$ that is consistent with T such that $T \cup S \models g$ the (T, A, g)* synthesis problem.

Our thrust in this section is to show that we can use *default* assumptions to focus this process. Thus in planning our trip from Stanford to MIT, we generated the incomplete plan of getting to San Francisco airport (SFO) and thence to MIT via Logan; we *assumed* that the intermediate steps of getting from Stanford to SFO, from SFO to Logan, and from Logan to MIT were all things that we could accomplish if we considered the problem in more detail.

Let us suppose for the moment that we are actually in a position to prove rigorously that we can get from Stanford to MIT via the airports suggested. (In practice, of course, all sorts of things can go wrong, and we must leave many small details of the trip unplanned until we actually encounter them.) If this is the case, then our decision to go via SFO and Logan really has done little more than focus our search effort – the detailed plan could have been generated initially instead.

What we have done is to augment our assumption set A with a set of default assumptions D (in this example, the assumptions that we can get from Stanford to SFO and so on). Then, having solved the planning problem given the default assumptions, we attempt to eliminate these assumptions by showing that they are in fact provable from our original set A after all. Here, then, is the procedure for solving the (T, A, g) synthesis problem using the default set D:

Procedure 2.3 *To solve the (T, A, g) synthesis problem:*

1. *Solve the $(T, A \cup D, g)$ synthesis problem to obtain a solution $\{a_1, \ldots, a_m, d_1, \ldots, d_n\}$.*

2. *Set $A_0 = \{a_i\}$ and $T_0 = T \cup A_0$ and solve the (T_0, A, d_1) problem to obtain A_1.*

3. *Now set $T_1 = T_0 \cup A_1$ and solve (T_1, A, d_2) to obtain A_2. Repeat to eventually solve (T_{n-1}, A, d_n) to obtain A_n.*

4. *Return $\cup A_i$ as a solution to the original problem.*

This procedure works by first solving the original problem using the defaults, and then replacing the defaults one at a time by sets of nondefault assumptions that entail them. As we have described it, we are assuming that it is always possible to replace the defaults in this fashion, so that they are all consequences of information elsewhere in the database; in practice, this may not be the case. In the next section, we will modify the above procedure to cater to this possible difficulty.

Before proceeding, however, we should point out that the idea of using defaults to represent assumptions involved in solving synthesis problems is not a new one. The connection between ATMS's and nonmonotonic reasoning is well-understood and has been described by a variety of authors [34, 81, and others]; the THEORIST system is built around the idea that defaults represent hypotheses that can be used in problem solving [76]. Our contribution here is not in identifying the connection between nonmonotonic reasoning and hypothetical solutions to synthesis problems, but in proposing that these hypothetical solutions can be used to focus the search for more certain results.

2.2.2 Computational value

Whether or not using defaults to focus search is a good idea from a computational point of view depends upon a variety of assumptions about the set of defaults D: Can these defaults really be used effectively to shorten the length of the original proof? Are the defaults likely to point the way to a solution to the original synthesis problem, or are they likely to be blind alleys? And so on.

In order to give a quantitative estimate of the time Procedure 2.3 needs to solve a particular query, we will need to make a variety of assumptions about these issues. Specifically, we assume that:

1. The relative cardinalities of the sets D and A is given by λ. We take

$$\lambda = \frac{|D|}{|A|}$$

   the ratio of the number of defaults to the number of original assumptions. We will go further and assume that since the number of assumptions in the default synthesis problem $(T, A \cup D, g)$ is $1 + \lambda$ times the number of assumptions in the original synthesis problem (T, A, g), the branching factor in the new problem is $(1 + \lambda)b$, where b is the branching factor in the original problem.

2. If S is a solution to some synthesis problem with branching factor b and there are m assumptions in S, then the time needed to solve the synthesis problem is of order $b^m = b^{|S|}$. This will be valid if b is the effective branching factor between additions to the set of assumptions being constructed as a solution to the problem.

3. Suppose that S is the smallest solution to a synthesis problem (T, A, g). Then there is some $n \geq 1$ such that the expected size of the smallest solution to the default synthesis problem $(T, A \cup D, g)$ is $|S|/n$.

   Roughly speaking, n here is the number of original assumptions that we expect to be covered by a single default; if $n = 1$, we do not expect the defaults to be useful in shortening the length of the solution.

4. There is some fixed l and probability p such that for an arbitrary default $d \in D$ and an arbitrary subset $A' \subseteq A$ consistent with T, p is the probability that the synthesis problem $(T \cup A', A, d)$ admits a solution of length l or less.

   We clearly expect $l \geq n$, since each default covers on average n base-level assumptions; if p is relatively large and l relatively small subject to this condition, each default is likely to expand into a fairly small number of nondefault assumptions. In general the probability that the $(T \cup A', A, d)$ synthesis problem admits a solution may vary depending on the sets A and A' and the particular default d; it suffices for the probability that this synthesis problem admits a solution to be no *less* than p.

The procedure that we will consider is a slight variant of Procedure 2.3. Here it is:

Procedure 2.4 *To solve the (T, A, g) synthesis problem:*

1. *Solve the $(T, A \cup D, g)$ synthesis problem to obtain a solution $\{a_1, \ldots, a_m, d_1, \ldots, d_n\}$.*

2. *Set $A_0 = \{a_i\}$ and $T_0 = T \cup A_0$ and attempt to solve the (T_0, A, d_1) problem to obtain A_1, requiring that $|A_1| \leq l$.*

3. *If no such A_1 exists, abandon the default solution and solve the original synthesis problem (T, A, g) directly.*

4. *Otherwise, set $T_1 = T_0 \cup A_1$ and attempt to solve (T_1, A, d_2) to obtain A_2 with $|A_2| \leq l$. Repeat to either eventually solve (T_{n-1}, A, d_n) to obtain A_n; if A_i doesn't exist for some i, solve (T, A, g) directly.*

5. *Return $\cup A_i$ as a solution to the original problem.*

As remarked at the end of the previous section, the reason that we use this modified procedure is that it is possible that the default solution found in step 1 does not expand into a full solution, and we need to bound the amount of time spent trying to expand the various defaults in subsequent steps.

Note also that if step 2 or 4 fails to find a solution using the extended assumption set $A \cup D$, we simply give up, as opposed to either backtracking to another default solution or attempting to modify the existing solution in a way that addresses the difficulty. Even given this extremely conservative behavior, however, Procedure 2.4 leads to computational savings:

Theorem 2.5 *Let (T, A, g) be a synthesis problem for which the shortest solution is of length k. Then provided that $l \ll k$, it will be more efficient to solve this synthesis problem using Procedure 2.4 than by conventional means whenever*[7]

$$p > \frac{1+\lambda}{b^{n-1}} \tag{5}$$

Theorem 2.5 is of substantial practical value. Consider, for example, the conservative assumptions that $n = 2$ (so that every default corresponds to only 2 nondefault assumptions) and $\lambda = 1$ (so that including the default assumptions doubles the size of the assumption set). Now (5) becomes

$$p > \frac{2}{b}$$

Since most synthesis problems have fairly large branching factors, this is a very weak restriction.

Theorem 2.5 can also be rewritten in the following form:

Corollary 2.6 *Procedure 2.4 can be expected to lead to computational savings whenever*

$$\frac{t(\mathit{default})}{t(\mathit{original})} < \mathit{prob}$$

where $t(\mathit{default})$ is the expected time needed to solve the default synthesis problem, $t(\mathit{original})$ is the time needed to solve the original one, and prob is the probability that the default solution can be expanded to a complete one.

Roughly speaking, it is worth risking an "investment" of 10% of the time needed to solve a particular problem if we have more than a 10% chance of solving the problem in this fashion. It is interesting to note that the cost of replacing the defaults in the tentative solution is sufficiently small that it does not enter into the above result.

2.2.3 Planning examples

The idea that default rules can be used to generate islands in a synthesis problem allows us to reinterpret the default information typically used by systems that reason about action. As an example, consider the frame axiom. This tells us that facts true in one situation will remain true in subsequent situations:

$$\mathtt{holds}(p, s) \land \neg\mathtt{ab}(a, p, s) \supset \mathtt{holds}(p, \mathtt{result}(a, s)). \tag{6}$$

Applied to planning, (6) suggests that when planning for a conjunctive goal, we simply achieve the subgoals individually. The frame axiom gives us reason to assume that previously

[7] Results in this section are proved in [36].

achieved subgoals will not be clobbered by subsequent actions. We made this point in Section 2.1, and it will continue to appear throughout this report.

Of course, the frame axiom is not valid in general. But much of the computational expense involved in planning is a consequence of the need to regress all of the goals through the various actions being considered. In situations where this regression is not needed, an optimistic approach will succeed much more quickly than the conventional one; gambling that this will be the case is worthwhile because the payoff is quite large if we are correct. Here is Elkan's observation again [20].

As another example, consider what might be called the "qualification assumption." By this we mean the assumption commonly made when reasoning about action that actions are unqualified in the absence of information to the contrary. Thus we assume that an attempt to start a car will succeed even if there is no information available explicitly stating that there is no potato in the car's tailpipe. Perhaps more realistically, attempts to drive the car are assumed to succeed even in the absence of specific information indicating that none of the tires is flat.

Applying this idea to planning leads us to assume that our intended actions are unqualified. This will typically be the case, and we will be saved the effort of worrying about potential qualifications as we construct the plan.

Two other advantages also come to mind here, although we will need to extend Procedure 2.4 in order to exploit them. The first involves cases where although the qualification assumptions hold (i.e., there is no potato in the tailpipe), we do not *know* that they hold. In this case, there may well be no solution to the original problem, since the necessary knowledge (about the state of the tailpipe) is simply missing from our database.

This is rather different from the previous ideas that we have considered. From a formal point of view, we are presented with incomplete domain information in the sense that our base set T is only a subset of the "complete" base set T'. Although we really want a solution to the (T', A, g) synthesis problem, we are forced to attempt to solve the (T, A, g) synthesis problem. Using the defaults, we solve the $(T, A \cup D, g)$ synthesis problem instead, and observe:

1. If the defaults used in the solution are all consequences of T' (as is the case with qualification assumptions), then the solution obtained will in fact be a solution to the (T', A, g) synthesis problem.

2. Even if not all of the defaults used are consequences of T', the solution can still focus our subsequent efforts in a fashion similar to that described in Section 2.2.2.

Finally, it may be the case that the qualification assumption actually fails – perhaps there is a potato in the tailpipe after all. In this case, the default solution to the synthesis problem can be expected to be similar to the solution to the problem itself (which presumably involves clearing the tailpipe and then proceeding as planned).

2.2.4 The consistency check

In the synthesis problems we have examined, we proceeded by first finding a default solution to the query, and then filling in the details of this skeletal solution. Thus instead of simply using a default rule to conclude that it is possible to fly from San Francisco to Boston, we can consult with a travel agent and confirm that it is indeed possible, and that American Airlines flies the route in question.

Default inference does not generally proceed in this fashion. Existing approaches [78] function first by finding a default solution to a problem, and then attempting to prove that the default solution *cannot* be extended to a certain one. Instead of finding a flight from San Francisco to Boston, we would attempt to show that there cannot be one. (Perhaps only Boeing 747's can fly transcontinentally, and the runways at Logan are too short for them to land.) When we fail to conclude that our default solution is *in*valid, we assume that it is in fact acceptable.

This is in contrast with the approach we have described, where we first draw a default conclusion and then attempt to prove it to be monotonically valid. But provided that our initial database is consistent, both approaches lead to the same result. Given that we can prove that our default conclusion is in fact valid, we do not need to attempt to prove it false. In our example, the existence of the American flight from San Francisco to Boston guarantees that we will fail in our attempt to prove that no such flight exists.

We have already remarked that in other instances it may be that there is no monotonic proof that we will be able to complete some portion of our skeletal plan. Consider the initial action of driving to the San Francisco airport: The highway may be closed, we may run out of gas, our car may be in the shop, and so on. Here, we may indeed take the alternative tack of considering possible proofs that we *can't* drive to the airport.

If the consistency check succeeds, so that we are unable to prove that there is *no* solution to our original problem that passes through the default island, we tentatively conclude that we will be able to complete the plan at some later date. Thus the final leg of our journey is left almost completely unspecified – I may have some vague notion that I'll rent a car when I get into Boston, but no idea about which car rental agency I'll use, how I'll find them, and so on. This is similar to the discussion of the qualification problem in the last section, where we conceded that it might be possible to find a solution to a synthesis problem given an incomplete description of the domain in question.

The consistency check may fail, however. Perhaps my car is in the shop and I can't drive to San Francisco airport or, as in the previous section, perhaps there is a potato in my tailpipe after all. Procedure 2.4 would now have us simply abandon our default plan and construct a monotonic replacement. More effective would obviously be to "debug" the default solution to the original synthesis problem. As we will see in Section 3.3, the approach we have outlined in this section is a useful first step in this direction.

Before turning to other issues, we should remark that there is another way in which we can understand our ideas generally. The reason that most approaches to nonmonotonic reasoning require a consistency check is that a declarative system is useless without it – there is simply no point in asserting a conclusion that is provably false.

In control problems, this is not the case. If it is possible to draw quick conclusions about the portion of a search space that should be expanded next, those conclusions have value even if subsequent consideration would show them to be false. The reason is that the potentially large time needed for this "subsequent consideration" may not be justified in terms of the time saved in the search itself. "Fast but not accurate" is unacceptable behavior for most declarative systems, but is *necessary* behavior in control work; because we are using the results of our default analysis primarily to control subsequent search, we are able to take advantage of default conclusions without explicitly checking their consistency.

2.3 Contrasting defaults and hierarchies

The overall methodology described in the previous section approaches planning problems as follows:

1. Assume by default that every high-level plan can be expanded into a low-level one.
2. Using this assumption, find a plan that succeeds by default in achieving the goal in question.
3. Prove that the default assumptions used are monotonic consequences of the database. A constructive proof here will be equivalent to fleshing out the high-level plan into lower-level actions.

In this section, we extend this idea in two ways. First, we show that every hierarchical system can be represented in this fashion, so that no expressive power is lost by viewing hierarchies as defaults. Second, we show that there is an analog to the ramification problem in this setting: If the planning hierarchy is situation-dependent, existing hierarchical planners will need to split actions into an exponential number of subtypes while a planner taking a declarative approach will not.

Before proceeding, however, we should be careful to stress that the claims we are making are representational only. This overall section of the report describes epistemological results, not heuristic ones. We will continue in Section 3 by developing a declarative planner that incorporates the ideas we have discussed.

Our intended contribution in this section is thus only to show that effective use of conventional planning hierarchies can require an exponentially large domain description, and that declarative methods can be used to overcome this problem. These arguments add further weight to the view expressed in the introduction and in Section 3.2 that practical planners will eventually need to be declarative systems.

In Section 2.3.1, we show that defaults can be used to describe existing planning hierarchies; in Section 2.3.2, we go on to show that the default approach is better able to handle hierarchies that are situation-dependent. More general remarks are contained in Section 2.3.3.

Action	Preconditions	Effects

go-through(d)	in($r1$)
	door($d, r1, r2$)
	open(d)
	next-to(d)
		¬in($r1$)
		in($r2$)

make-open(d)	in($r1$)
	next-to(d)
	door($d, r1, r2$)
		open(d)

go-to(obj)	location(obj, r)
	in(r)
		next-to(obj)

Table 1: Primitive actions of the robot

2.3.1 Hierarchical planning achieved with defaults

We now introduce a simple planning problem that is best solved hierarchically. We present a typical hierarchical solution and then show that identical behavior can be obtained using defaults. We go on to show that the technique used is completely general, proving that a default description of hierarchy loses none of the computational benefits of hierarchical planning.

A sample domain Consider a domain with a robot wandering through a maze of rooms that are connected by doors. Within a room, the robot may be located near any object or door in that room. The primitive actions that the robot can take include walking from room to room, going to a location within a room, and opening a door. The task of the planner is to generate plans for the robot to navigate through the maze to get to some particular object.

The domain actions appear in Table 1. The robot can walk through any door it is next to, provided that the door is open. It can open any door it is next to, and go to any object located in its current room.[8] The predicate in identifies the room in which the robot is currently located, and next-to identifies objects it is adjacent to.

It is clear that when planning in this domain, the robot's location within a particular room and the question of whether or not a door is open should be deferred while a high-level plan is being developed; the reason is that we can always expect to be able to develop a plan to either open a door or move the robot next to any object in its current room. The natural hierarchy in this domain is therefore one where we plan the go-through actions first; let us suppose that we next plan the make-open actions, and finally the go-to actions. It is well known that this hierarchical view of the problem results in an exponential reduction in the

[8] The axiomatization in Table 1 fails to indicate that going to an object x should delete any existing next-to facts. This is an instance of the *ramification problem* [26], and can be treated in a variety of ways. The arguments we will present are independent of the treatment used.

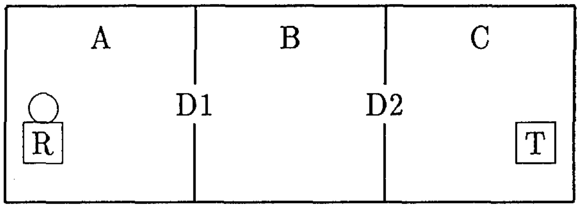


Figure 1: The robot's environment

Abstraction level	Plan	Truth value
1	`through(D1), through(D2), go-to(table)`	d^2t
2	`open(D1), through(D1), open(D2), through(D2), go-to(table)`	dt
3	`go-to(D1), open(D1), through(D1), go-to(D2), open(D2), through(D2), go-to(table)`	t

Table 2: Refining plans

size of the search space [61], and this reduction is achieved by hierarchical planning systems such as SIPE [106] or ABSTRIPS [83] (not to mention many others).

Now suppose that our robot's environment is as shown in Figure 1, consisting of three rooms, `A`, `B` and `C`, connected by doors `D1` (between `A` and `B`) and `D2` (between `B` and `C`). The robot begins in room `A` with all of the doors shut and the goal of standing next to a table in room `C`. A hierarchical planner using the hierarchy described above would generate the plans shown in Table 2. (We have abbreviated `go-through` to `through` and `make-open` to `open` in the table.)

In existing hierarchical planners, this decomposition of the problem is achieved using a special operator representation. In ABSTRIPS, for example, each subgoal has an associated criticality value that indicates how long that subgoal should be deferred.

Treatment using defaults Base-level default knowledge can reproduce this hierarchical decomposition without resorting to the use of a specialized representation language. In the above example, we simply provide information indicating that certain subgoals are likely to be easily achieved:

$$\begin{array}{rl} \texttt{achievable(open(}x\texttt{))} & [d^2t] \\ \texttt{achievable(next-to(}d\texttt{))} & [dt] \end{array} \tag{7}$$

Our notation here is that dt corresponds to a first-order default, d^2t to a second-order default, and so on. The commonsense meaning of the first default in (7) is that we can open any door, and the second means that we can easily get to any object in our current room. The different default values of dt and d^2t are used to indicate that we are more certain of our ability to navigate around a particular room than to open a door.

To use the information in (7), suppose that the form of the axiom telling us when an action succeeds is as follows:

$$\texttt{occurs}(a) \wedge [\forall p.\texttt{prec}(p,a) \supset \texttt{sat}(p)] \supset \texttt{succeeds}(a)$$

$$\texttt{holds}(p) \supset \texttt{sat}(p) \qquad (8)$$

Different declarative systems will obviously represent actions differently, but this seems a reasonable choice. An action succeeds if it occurs and its preconditions are satisfied; a specific precondition is satisfied if it holds in the situation in question. This description is similar to the one we will use in Section 3.

We can now modify (8) to become the following:

$$\texttt{holds}(p) \vee \texttt{achievable}(p) \supset \texttt{sat}(p) \qquad (9)$$

As we will see in Proposition 2.7, this has exactly the desired effect of capturing the hierarchy in our declarative setting.

Both of the defaults in (7) are base-level information about the planning domain and can now be used to control the planner's search. Specifically, the search for a plan that has truth value d^2t can essentially ignore the question of whether doors are open and where in the room a particular object is; the subsequent search for a refined plan with truth value dt will ignore only the question of where objects are within rooms. Somewhat more specifically, plan 1 in Table 2 has truth value d^2t, plan 2 has truth value dt and plan 3, which is guaranteed to succeed, has truth value t.

The two default statements cause the declarative planner to generate partial plans that are identical to the ones generated by a traditional hierarchical planner. We can also mechanize the process of turning an ABSTRIPS hierarchy into a set of defaults:

In ABSTRIPS, relations are assigned priorities; when solving the problem, the relations are considered in order of decreasing priority. In the example we are considering, the open predicate would be assigned a value of 2 and the `next-to` predicate a value of 1. In other words, the initial plan can ignore all instances of open or `next-to` as preconditions; the first refinement can ignore instances of `next-to`.

Proposition 2.7 *Suppose that an* ABSTRIPS *system labels each of its predicates p_i with a priority v_i. Then the functioning of* ABSTRIPS *can be duplicated using the defaults*[9]

$$\texttt{achievable}(p_i) \quad [d^{v_i}t] \qquad (10)$$

[9] For proofs, see [47].

This proposition shows that we can always continue to enjoy the computational benefits of hierarchical planning in a declarative setting.[10]

Deducing control information from base-level knowledge about the domain (default statements about preconditions) allows us to select our hierarchy automatically. SIPE and NONLIN, for instance, do not address the problem of classifying operator preconditions into filters and subgoals (Wilkins calls them *preconditions* and *subgoals* [106]). Filters are preconditions to an action which the planner should not, for heuristic reasons, plan to achieve. Subgoals are preconditions that the planner should try actively to achieve. The classification of preconditions into filters and subgoals in these classical planners is generally left for the user to solve in an ad hoc manner, using a hierarchy or in some other fashion.

Given base-level default information about the achievability of certain predicates, the deferrable preconditions of an operator become precisely those for which achievability can be proven by default. The problem of distinguishing filters from subgoals thus becomes one of correctly stating the base-level assertions. Provided that the default assertions fairly represent our knowledge about the world, we expect that the induced hierarchy of actions will also be correct. We made similar claims for the derivation of control knowledge generally in Section 2.1.

2.3.2 Beyond simple hierarchies

We now extend our example in a way that reveals some limitations of conventional hierarchical descriptions, and go on to show that default information can be used to avoid these limitations.

Extending the example We complicate the world of the robot a bit, adding cabinets in every room and specifying that doors have locks on them. Inside the cabinets are keys that unlock the doors, although there is no guarantee that the cabinet in any particular room will contain the keys to that room's doors.

The actions in our extended domain appear in Table 3. We can unlock any door we are next to, if we have the key. We can take a key if it is in a cabinet that we are next to. The problem of opening a door d now becomes much more subtle. If d is locked, we need to find a path through unlocked doors to the cabinet that contains d's key.

The failure of static hierarchies Consider the problem of writing a set of hierarchical operators corresponding to the primitive actions available to our robot. It is clear that we must consider the type predicates `key` and `door` to be of the highest criticality, since they are unchanged by any actions we might perform. With a little thought, it becomes clear that the `in` precondition of the `go-through` operator clearly should also be planned for early.

[10] In keeping with the discussion of the previous section, we are assuming that it is possible to use our declarative system to draw default conclusions *without* performing the consistency check required by most formalizations of nonmonotonic reasoning.

Action	Preconditions	Effects

go-through(d)	in($r1$)
	door($d,r1,r2$)
	open(d)
	next-to(d)
		¬in($r1$)
		in($r2$)

make-open(d)	in($r1$)
	next-to(d)
	door($d,r1,r2$)
	unlocked(d)
		open(d)

go-to(x)	location(x,r)
	in(r)
		¬next-to(y)

unlock(d)	key(k,d)
	holding(k)
	next-to(d)
		unlocked(d)

take(k)	cabinet(k,c)
	next-to(c)
		holding(k)
		¬cabinet(k,c)

Table 3: Actions in the extended robot world

What about the open precondition of the go-through operator? If we continue to let open have a lower priority than in, the openness of doors will only be considered after planning a complete route from start to finish. However, this ordering is likely to result in an abstract plan that we will be unable to expand. In expanding a plan that involves opening a locked door for which we have no key, the planner will generate a subplan for the robot to go to another room, retrieve the key and come back to the first locked door. This subplan will set off a cascade of sub-subplans for retrieving the keys to the locked doors that the robot runs into while on its first subtask and so on; if the planner succeeds in solving its problem this way, it will be by blind luck more than anything else. Any plan found is almost guaranteed to be disastrously baroque.

Note that this problem arises if the planner even *once* makes an incorrect assumption that a door will be easily opened. This is obviously quite likely for any plan that takes us through many doors.

Solutions in the static model If we are committed to the conventional description of hierarchies, we can salvage it in one of two ways. We can abandon the hierarchy altogether, or split one of our operators into two.

Abandon By giving open a high criticality value, we can choose to never defer it. Our abstract plans will no longer suffer from the "cascade" effect, but abandoning the hierarchy will cause us to lose the computational benefits of hierarchical planning. This seems not to be a viable solution for computational reasons, especially if most of the doors are unlocked.

Split The other option is to split the go-through operator into two. One operator tells how to walk through easy doors; the other tells how to walk through hard doors. Assuming that our representation allows it, go-through-easy-door will give open a low criticality value, and will have filters making the rule applicable only if the door is unlocked or a key can easily be retrieved. go-through-hard-door will give open a high criticality value and will have filters guaranteeing that it is only applied when the door is in fact difficult to open. This will guarantee that we only defer open when it really is deferrable, avoiding the problem introduced earlier. An alternative but equivalent split creates the operators unlock-with-key, which unlocks doors for which we have the key or can get it easily, and unlock-without-key, which deals with the more difficult situation.

But now suppose that we have another precondition to go-through that is also sometimes deferrable and sometimes not. Perhaps it is easy to get next to most doors, but for some next-to is a problem because there are obstacles obstructing the path to the door; moving the obstacles might require retrieving tools from other rooms. We now have the same problem with next-to that we did earlier with open: If we always defer next-to, a new cascade of subgoals makes it unlikely that we solve our original problem. If we never defer next-to, we lose the computational advantages of hierarchy. Sometimes deferring next-to forces us to split the go-through action into two again. Since we have already split this action along the easy-door/hard-door axis, the new split will result in four subactions. The following result is immediate:

Proposition 2.8 *If an action has n preconditions, each of which is deferrable in some situations but not in others, a static hierarchy may need to split the action into 2^n subtypes in order to plan effectively.*

Note that it is not only the size of the domain description that is growing exponentially here; by introducing more actions we are also increasing the planner's branching factor from n to 2^n. We see from this that splitting the operators used by conventional hierarchical planners is not an attractive option in general.

Solution in the declarative model In the declarative approach, solving this problem is trivial. If a predicate p_i should be given priority v_i *but only in the situation* s_i, we need merely replace (10) with

$$s_i \supset \texttt{achievable}(p_i) \quad [d^{v_i}t] \tag{11}$$

The s_i's appearing here will refer to various features of our domain; one instance of (11) might be

$$\texttt{holding}(k) \wedge \texttt{key}(k,d) \supset \texttt{achievable}(\texttt{unlocked}(d))$$

(with truth value d^jt for some j), saying that we don't have to worry about unlocking a door for which we already have the key. Given this axiom, note that we will assume that we can unlock d if there are other actions, also succeeding with truth value d^jt or better, that manage to establish s_i. So if we continue to make next-to the least critical of our preconditions, we can easily plan to collect keys and use them before worrying about the details of approaching the cabinets in which specific keys are contained.

Situation-dependent hierarchies arise in other settings as well. Consider the original axiomatization of our robot's world once again, where we indicated that opening a door d had three preconditions: that the robot be in the same room as d, that it be next to d, and that d be a door. The first of these preconditions is in some sense irrelevant, since the robot can't be next to d unless it is in the same room as d is.

Irrelevant though it may be from a declarative point of view, the precondition is crucial to a hierarchical planner working in this domain. To see why, recall that when constructing a plan that succeeds only by default, we ignore any precondition involving next-to. When we do this, the make-open action will have only a single precondition – that the object being opened be a door.

The result is that the robot, trying to get to the table in Figure 1, may form the plan of first opening both doors and only then walking to B, C and the table! By including the dummy in precondition to the make-open action, we ensure that the robot, when constructing the default plan, cannot expect to open a door d unless it is in the same room as d is.

The declarative approach provides us a much cleaner way of achieving this affect. Instead of saying that

$$\texttt{achievable}(\texttt{next-to}(d))$$

is true by default, we say that

$$\texttt{location}(d,r) \wedge \texttt{in}(r) \supset \texttt{achievable}(\texttt{next-to}(d)) \tag{12}$$

is true by default. This really captures the sense of the default: We can maneuver next to any object in our current room, but not necessarily next to other objects in the domain.

There are a variety of advantages to this formulation. First, existing methods require us to add a dummy in precondition to every action that has a next-to precondition; it is obviously simpler to merely replace (7) with (12).

Second, there are domains that are awkward to describe using the dummy preconditions needed by static hierarchies. Imagine that our domain contains some invisible objects, and the default we want to introduce says that we can only maneuver next to *visible* objects in our current room.

If we want to avoid splitting our actions into subtypes depending on the visibility of the door or other objects involved, we will need to introduce a dummy visible precondition. But this is unfaithful to the sense of the original domain description, which permitted us to open invisible doors if we could somehow manage to maneuver next to them.

These collected observations suggest another way of looking at Proposition 2.8. What we have seen is that the question of the criticality or priority to be assigned to a precondition is one that requires us to do some reasoning. How hard is it to open a door or get next to it? A static hierarchy such as that used by existing planning systems cannot do this kind of reasoning, and must determine the priorities by lookup alone.

Of course, there is a standard way to ensure that no reasoning is needed to solve a problem: Instead of working with the existing axiomatization, use its deductive closure. This guarantees that no inference will be needed in determining the priority levels, but the deductive closure of a database consisting of sentences like (11) can be expected to be of size exponential in the size of the original database. This is why the number of action subtypes grows exponentially with the number of split preconditions.

In some cases, of course, computing the deductive closure of a declarative database can save a great deal of effort at runtime; it might be very difficult to backward chain through rules like (11) to determine what priority should be assigned to a predicate in some particular situation. This seems unlikely to be true in planning examples, however, since the priority determination can be expected to depend in a fairly straightforward way on the values taken by situation-dependent variables (is the door locked?). In those cases where significant inference is needed to get from situational information to criticality values, we can still choose to forward-chain on the rules being used without splitting our actions into a catastrophic number of subtypes.

2.3.3 Other descriptions of hierarchy

Lifschitz has shown [63] that any system making the STRIPS assumption will have difficulty dealing with the ramification problem. Wilkins acknowledges that such a system cannot be powerful enough to encode complex control knowledge: "Systems such as SIPE that do hierarchical planning can use abstract operators to encode some metaplanning knowledge, but ... the most interesting metaplanning ideas cannot be encoded in this manner" [105]. In this section, we have confirmed Wilkins' statement, shown that the problem is closely related to the need to split STRIPS operators into subtypes to solve the ramification problem, and proposed a way to use base-level default information to overcome these difficulties. Atomic defaults can be used to capture existing hierarchical notions, while compound default sentences can be used to represent hierarchies that have previously eluded formulation.

Other authors have also considered these issues, although their work has a flavor rather different from ours. Feldman and Morris discuss the distinction between filters and subgoals [24], but their analysis focuses on the detection of loops in planning search and leaves the question of hierarchy unaddressed. Knoblock discusses the automatic derivation of a planning hierarchy [58], but the methods used are primarily syntactic and are concerned with the development of a static hierarchy in any event. Smith and Peot suggest that Knoblock's analysis should focus on more semantic features [97] but they, too, limit their attention to static hierarchies.

This is to be expected; it seems impossible to incorporate dynamic hierarchies into nondeclarative planners. Although it is true that declarative planning has problems of its own, the analysis we have presented shows that it also has important capabilities that nondeclarative methods lack. It is to the problems – and their solutions – that we now turn.

3 Heuristic results: Planning

The epistemological arguments of the previous section were in some sense the easy part. Now we turn to the difficulties involved in bringing our observations into practice. In the next three subsections, we present methods by which a planner can be made interruptible (Section 3.1), declarative (Section 3.2) and able to deal with subgoals separately (Section 3.3).

3.1 Interrupting declarative systems

Our first aim is to discuss the difficulties that arise if declarative methods are to be used in real-time or other systems that provide only limited computational resources. Conventional declarative systems are inappropriate for use in these applications because of the unpredictable amount of time needed for them to respond to queries. The two solutions that have been proposed are to abandon declarativism (as suggested in the work on situated automata theory and its variants [1, 16, 56, 72, 82, 84]) or to restrict the declarative language in a way that guarantees that inference is tractable [22, 87, and many others].

There are problems with both of these approaches. The arguments in favor of declarativism are strong enough that abandoning the methodology should surely be a last resort; the restricted languages considered by other authors are often too inexpressive to be applied to problems of practical interest. We saw the importance of declarative formalisms in the control of reasoning in Section 2, and will discuss the heuristic value of these formalisms in Section 3.2 as well.

We will modify the declarative methodology so that declarative systems can be applied to real-time problems; somewhat more precisely, we will present a declarative approach to inference that is what Dean and Boddy have called an *anytime* algorithm. The associated computation can be terminated at any time and an answer returned, and the quality of these answers improves in some well-behaved manner as a function of time (paraphrased from [11, page 52]). No declarative system has these properties: Interrupt a first-order theorem prover and the most useful information you are likely to recover is the current fringe of the proof tree.

We argue that modality provides the answer. Notions of groundedness that have arisen in work on nonmonotonic reasoning [59, 60, 70] suggest that one of the purposes of nonmonotonic modal operators in declarative systems is to insulate their arguments from other information in the database, so that the truth or falsity of a modal expression can be determined solely by an analysis of the sentential arguments it contains. We will develop from this the view that modal operators are markers for points at which inference can be suspended and partial answers returned.

As an example, consider the following story:

> Fred fell out of an airplane over a farmer's field. The good news is that he was wearing a parachute. The bad news is that the parachute didn't open. The good news is that there was a haystack in the field. The bad news is that there was a

```
dead(X) :- falling-onto(X,Y), not(good-parachute(X)), not(soft(Y)).
good-parachute(X) :- wearing-parachute(X,P), not(broken(P)).
soft(Y) :- haystack(Y,H), not(ouch(H)), not(missed(H)).
ouch(H) :- pitchfork(H,P), not(missed(P)).

falling-onto(fred,field).
wearing-parachute(fred,parachute).
broken(parachute).
haystack(field,haystack).
pitchfork(haystack,pitchfork).
missed(pitchfork).
missed(haystack).
```

Figure 2: Fred's voyage

> pitchfork in the haystack. The good news is that Fred missed the pitchfork. The bad news is that he also missed the haystack.

As we read this, we flip back and forth between conclusions about Fred's eventual well-being. We begin by concluding that he will be killed, using a nonmonotonic rule that people who fall out of airplanes typically die. The presence of the parachute gives us additional information that we incorporate to conclude that the poor fellow is probably OK after all; then we learn that the parachute didn't open, and so on.

In this particular case, the alternating conclusions reflect changes in our information about the situation. But there are examples of similar behavior where we simply haven't taken the time to analyze the information that we do have. If you've ever walked away from your car and left the lights on, you know what we're referring to.

If we were to write a PROLOG program for leaving a car, part of it might look like this:

```
ok-to-leave(Car) :- keys(Car,K), in-pocket(K), lights-out(Car).
lights-out(Car) :- daytime, not(abnormal).
abnormal :- recent-rainstorm.
abnormal :- drove-through-tunnel.
lights-out(Car) :- observed-lights-out(Car).
```

It's OK to leave the car if we've got the keys in our pocket and the lights are out; the lights are typically out if it's daytime. But not if there was a recent rainstorm or we just drove through a tunnel; in these cases we actually have to check that the lights are out.

The problem is that it is all too easy to be sloppy about assigning a value to `abnormal`; we know from long experience that things are usually normal, so we are tempted to assume that `not(abnormal)` is true and we wind up returning to a car with a dead battery.

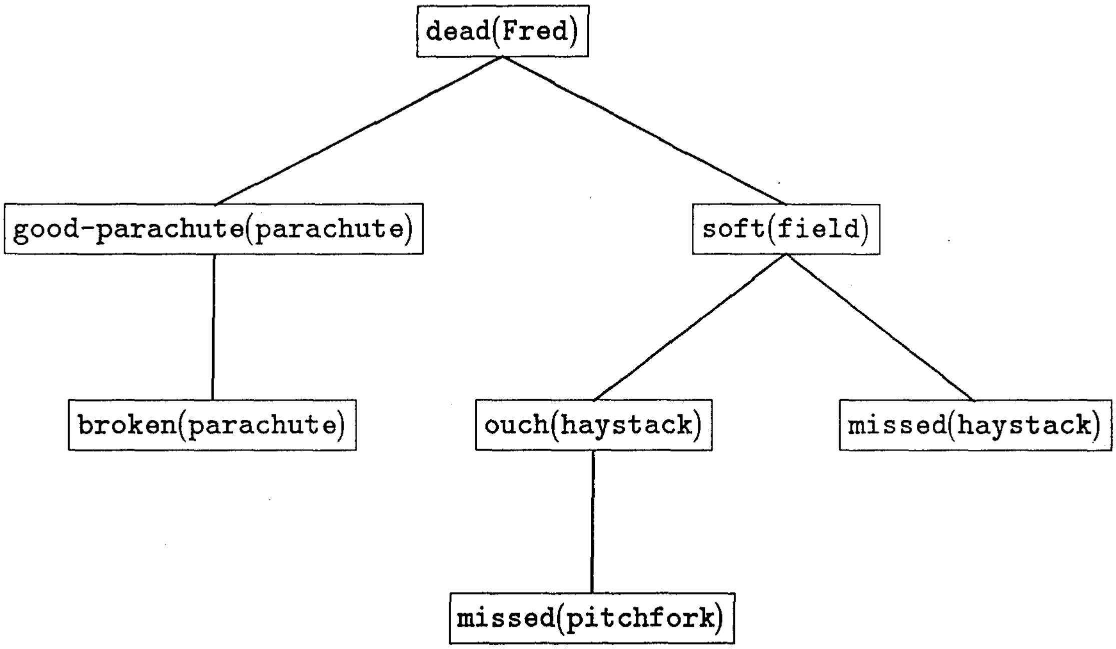


Figure 3: Modality as recursive descent

From a formal point of view, the example about the parachutist is no different; an axiomatization might look like the one in Figure 2. Writing it this way doesn't do much for the joke, but makes clear the parallels between the two examples: If we are sloppy in our analysis of a clause such as `not(missed(haystack))`, we will conclude that this clause is true, and therefore that `soft(field)` is true and Fred survives the fall. Sloppiness regarding `not(soft(field))` will lead us to conclude correctly that Fred dies, although for the wrong reason. PROLOG's negation-as-failure operator serves to mark those points where we might be sloppy in this way; put more positively, the operator marks points at which the inference process can be suspended and a partial answer returned. More thought might alter the computed conclusion, but we may be unable or unwilling to devote the computational resources required.

We can think of this in terms of a recursive descent through a dependency graph corresponding to PROLOG's reasoning. Since we know that Fred is falling toward the field, we can decide if he will die by determining whether he's wearing a functioning parachute and whether the field is soft. A recursive analysis of `good-parachute(Fred)` leads us to the goal of `broken(parachute)`; the recursion terminates here because `broken(parachute)` appears explicitly in the database. A full diagram of the recursive tree to which this process leads appears in Figure 3.

We will generalize this recursive descent model in two ways:

1. We will change the interpretation of the negation in Figure 2 from "negation as failure to prove" to "negation as failure to prove *so far*." By doing this, we will become able to respond to queries even if the subgoals they generate have not yet been analyzed

completely.

2. We need not restrict these ideas to negation as failure; any other operator that takes one or more sentential arguments can be treated similarly, provided that we can assign a value to the operator both before and after the investigation of the associated subgoals. A truth-functional semantics for modality such as that based on bilattices [35] will allow us to do this.

As an example, the translation mechanism that we will introduce shortly will replace the first PROLOG axiom in Figure 2 with

$$\texttt{falling-onto}(x,y) \wedge \neg\Box\texttt{good-parachute}(x) \wedge \neg\Box\texttt{soft}(y) \supset \texttt{dead}(x)$$

where $\Box$ is a modal operator of necessity, serving to indicate that we are able to prove the argument from other information in the database. In this purely declarative formulation, $\Box$ continues to mark points at which new inference processes are spawned; these new processes analyze the argument to any particular appearance of $\Box$ itself. In terms of the recursive descent description, the modal operators mark points at which new nodes are formed. In terms of anytime reasoning, they mark points at which inference can be interrupted and an approximate answer computed.

The remainder of this section makes these ideas precise. An outline is as follows:

1. In Section 3.1.1, we begin by describing the existing results underlying this work. We present a semantics for modal operators that allows us to evaluate them both before and after investigating their sentential arguments; the essential idea involves viewing the modalities truth-functionally [35].

2. In Sections 3.1.2 and 3.1.3, we use these results to describe an anytime procedure for computing the truth value assigned to a query. Approximate answers are computed quickly and the sequence of such answers converges to the correct one in the large-runtime limit.

3. In Section 3.1.4, we show that the resulting procedure generalizes existing work on stratification of logic programs [2, 30, 77, 103, 104], negative subgoals containing unbound variables [6, 7, 38] and some informal remarks of Perlis' [75]. Applications to planning appear in Section 3.2.

3.1.1 Preliminaries and previous work

Bilattices We have already remarked that our ideas depend on an ability to evaluate modal expressions both before and after considering their sentential arguments; this requirement is not supported by views of modality based on Kripke's possible world semantics [62]. Instead, we will use a *truth-functional* semantics for modality; the value assigned to a modal expression $m(p_1,\ldots,p_n)$ will be computed in terms of the values assigned to the sentential arguments p_1 through p_n. We can assign a preliminary value to the p_i by testing for their explicit appearance in the database; if a more complete analysis leads to a different value for some particular p_j, the truth-functional semantics assigned to m will lead to us to change the value of $m(p_1, \ldots, p_n)$.

As an example of a modal operator that admits a natural truth-functional semantics, consider the modal operator of necessity $\Box$. A natural interpretation for $\Box$ is given by

$$\Box(x) = \begin{cases} t, & \text{if } x = t; \\ f, & \text{if } x = f \text{ or } x = u \end{cases} \tag{13}$$

so that $\Box p$ is true if p itself is assigned the truth value true, and $\Box p$ is false if p is either false or unknown (i.e., those situations in which p is not necessarily true).

Note that the description given by (13) makes explicit use of the label u for a sentence that is not necessarily either true or false. This is a general property of our approach; if modal operators are to be assigned a truth-functional semantics, the truth values assigned to their sentential arguments must be taken from a set larger than the simple two-point set $\{t, f\}$.

Taking the values assigned to sentences from a set known as a *bilattice* [33] leads to a truth-functional description of modality that generalizes the conventional one [35]. This is the approach we take here: Truth values are elements of a bilattice, and modal operators are viewed truth-functionally. Since the technical results we will present depend critically on the details of the bilattice interpretation of modality, we begin with a brief review of bilattice theory, summarizing material that has appeared elsewhere [33, 35]. Bear in mind that our interest in bilattices is only in that they provide a convenient formal framework in which modality can be viewed truth-functionally; results similar to ours could be developed in any other formal system that also treated modality truth-functionally.

Definition 3.1 *A* bilattice *is a sextuple* $(B, \wedge, \vee, \cdot, +, \neg)$ *such that:*

1. $(B, \wedge, \vee)$ *and* $(B, \cdot, +)$ *are both complete lattices, and*
2. $\neg : B \rightarrow B$ *is a mapping such that:*
   (a) $\neg^2 = 1$, *and*
   (b) $\neg$ *is a lattice homomorphism from* $(B, \wedge, \vee)$ *to* $(B, \vee, \wedge)$ *and from* $(B, \cdot, +)$ *to itself.*

Informally, a bilattice is a set of truth values equipped with two partial orders $\leq_t$ and $\leq_k$, corresponding to the partial orders associated with the lattices $(B, \wedge, \vee)$ and $(B, \cdot, +)$ respectively. The $\leq_t$ partial order indicates how true or false a particular value is, with f being the minimal element of this partial order (least true) and t the maximal element (most true). The $\leq_k$ partial order reflects how much is known about a particular sentence; the minimal element here is u (completely unknown) and the maximal element is $\bot$ (representing a contradictory state of knowledge where a sentence is known to be both true *and* false).[11]

[11] Other authors have used $\top$ for this contradictory truth value, since it is the maximal element under the k partial order [27]. We use $\bot$ in part to conform to AI usage of $\bot$ for a contradictory sentence, and in part to avoid any typographical confusion with T.

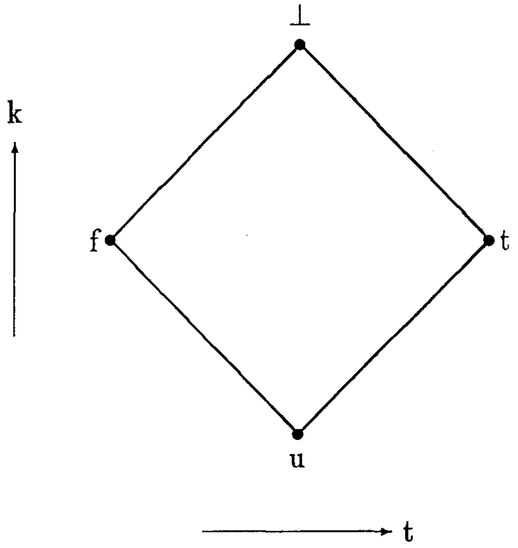


Figure 4: F, the smallest nontrivial bilattice

The least upper bound and greatest lower bound operators for the $\leq_t$ partial order are $\vee$ and $\wedge$ respectively, and correspond to the usual logical notions. The operators for the $\leq_k$ partial order are denoted by $+$ and $\cdot$. The operator $+$ corresponds to the combination of evidence from separate lines of reasoning to a single conclusion, while $\cdot$ is a "consensus" operator that indicates the strongest conclusion that can still be strengthened further to each of a variety of values.

Finally, every bilattice is equipped with a negation operator $\neg$ that inverts the sense of the $\leq_t$ partial order (corresponding to de Morgan's laws) while leaving the $\leq_k$ partial order unchanged (since if you know nothing about a sentence p, you also know nothing about its negation $\neg p$). This corresponds to condition (2b) of the formal definition, where $\neg$ is required to reverse the order of $\wedge$ and $\vee$ while retaining the order of $\cdot$ and $+$.

The smallest bilattice is the trivial one, consisting of a single point and for which all of the bilattice operations map that point to itself. The smallest nontrivial bilattice appears in Figure 4 and corresponds to first-order logic. This bilattice F consists of only the four truth values $\{t, f, u, \perp\}$; as mentioned earlier, $\perp = t + f$ is used to label a sentence about which there is contradictory information. The two partial orders are as labelled in the figure. Although $t >_t f$, they are incomparable with respect to the $\geq_k$ partial order, since neither represents a more informed state than the other.

Larger bilattices contain other truth values; an example is shown in Figure 5. This is the bilattice corresponding to default logic and includes the values of dt (true by default), df (false by default) and $*$ (both true and false by default, as in the Nixon diamond [80]). The fact that $f >_k dt$ in this bilattice reflects the fact that additional knowledge can cause a change of heart about a default conclusion.

Within this framework, we now make the following definition:

Definition 3.2 *A* truth assignment *is a function $\phi : L \to B$, where L is our declarative language and B is a bilattice.*

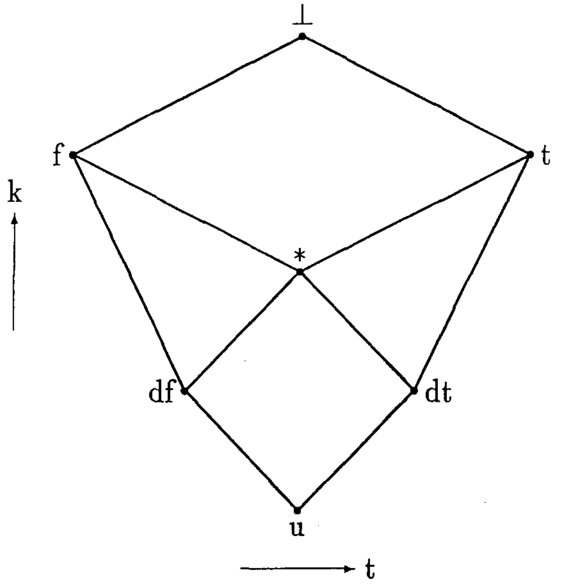


Figure 5: D, the bilattice corresponding to default logic

A truth assignment assigns a truth value (possibly unknown) to each sentence in our declarative language. We will often refer to the sentences labelled with truth values other than unknown as the *database.*

Note that truth assignments assign truth values to all of the sentences in our language, not just the atomic ones. Thus we are free in principal to say that p and $\neg p$ are both true; although the consequences of such a truth assignment are contradictory, it is not disallowed by Definition 3.2 itself.

Definition 3.3 *For truth assignments ϕ and ψ, we will write $\psi \geq_k \phi$ to mean that $\psi(p) \geq_k \phi(p)$ for every sentence $p \in L$ and $\psi \geq_t \phi$ to mean that $\psi(p) \geq_t \phi(p)$ for every sentence $p \in L$. If $\psi \geq_k \phi$, we will say that ψ is an* extension *of ϕ; if $\psi \geq_t \phi$ as well, we will say that ψ is a* positive extension *of ϕ.*

If ψ is an extension of ϕ, it means that the state of belief corresponding to ψ is uniformly "more informed" than the state of belief corresponding to ϕ. If ψ is a *positive* extension of ϕ, it means that all of the sentences about which additional information is available are more true under ψ than under ϕ.

The semantics of a bilattice system are given by a definition of *closure.* If ϕ is a truth assignment labelling each sentence p with explicit information about p, the closure of ϕ, $\text{cl}(\phi)$, should be a truth assignment that labels p with the conclusions that this information entails. As an example, if ϕ labels both q and $q \supset r$ as true, then $\text{cl}(\phi)$ should also label r as true since this is implied by the information in ϕ itself.

The formal definition of closure in this setting is given elsewhere [33]; our aim here is to give a vaguely constructive definition that appears in the earlier work only as a theorem. To understand the result we are about to present, suppose that $q \in L$ is some sentence in our language. What truth value should be assigned to q by $\mathrm{cl}(\phi)$?

If $S \subset L$ is a set of sentences entailing q, then the truth value to be assigned to the conjunction of the elements of S is

$$\bigwedge_{p \in S} \mathrm{cl}(\phi)(p)$$

If we denote this value by s, there should be a contribution to the truth value of q given by $u \vee s$. The reason that we disjoin with u can be seen if the conjunction is false, so that $s = f$. There should be no impact on the truth value of q in this case, since we are simply unable to draw useful conclusions from the implication $S \supset q$. The disjunction of s with u achieves just this effect. The reason we say that $u \vee s$ is only a *contribution* to the truth value of $\mathrm{cl}(\phi)(q)$, and not the truth value of $\mathrm{cl}(\phi)(q)$ itself, is because there may be many sets S that imply q and we want to combine the conclusions that can be drawn using each of them.

These arguments suggest that we take

$$\mathrm{cl}(\phi)(q) = \sum_{S \models q} u \bigvee \left[\bigwedge_{p \in S} \mathrm{cl}(\phi)(p) \right] \tag{14}$$

Once again, the motivation for (14) is as follows: There is a contribution for each set S that entails q. For each such set, we can compute the truth value of the conjunction of the elements of S by conjoining the truth values assigned to these elements individually. Since this conjunction only entails q and is not equivalent to it, we disjoin with unknown and then sum the results as S is allowed to vary.

As an example, suppose that our truth assignment ϕ labels the following statements as other than unknown:

$$\begin{aligned}
\phi[\texttt{Quaker}(x) \wedge \neg \mathrm{ab}_r(x) \supset \texttt{pacifist}(x)] &= t & (15)\\
\phi[\texttt{Quaker(Nixon)}] &= t & (16)\\
\phi[\neg \mathrm{ab}_r(x)] &= dt & (17)\\
\phi[\texttt{hippie}(x) \supset \texttt{pacifist}(x)] &= t & (18)\\
\phi[\texttt{hippie(Nixon)}] &= f & (19)
\end{aligned}$$

Quakers are pacifists unless they are (religiously) abnormal; hippies are definitely pacifists. Nixon is a Quaker but not a hippie. This example parallels one due to Reiter and Criscuolo [80]. What truth value should be assigned to $\mathrm{cl}(\phi)[\texttt{pacifist(Nixon)}]$?

There are two proofs that Nixon is a pacifist using information in the database. One uses (15)–(17); the other uses (18) and (19). If we assume that $\mathrm{cl}(\phi)$ and ϕ agree on the sentences appearing explicitly in (15)–(19), the value to be assigned to $\mathrm{cl}(\phi)[\texttt{pacifist(Nixon)}]$ is given by (14) as

$$\mathrm{cl}(\phi)[\texttt{pacifist(Nixon)}] = [u \vee (t \wedge t \wedge dt)] + [u \vee (t \wedge f)]$$

$$\begin{aligned} &= [u \vee dt] + [u \vee f] \\ &= dt + u \\ &= dt \end{aligned}$$

The proof involving hippies has no impact on the overall truth value because Nixon is not known to be a hippie.

But now suppose we add the following to our database:

$$\begin{aligned} \phi[\texttt{Republican}(x) \wedge \neg\texttt{ab}_p(x) \supset \neg\texttt{pacifist}(x)] &= t && (20) \\ \phi[\texttt{Republican}(\texttt{Nixon})] &= t && (21) \\ \phi[\neg\texttt{ab}_p(x)] &= dt && (22) \end{aligned}$$

Republicans who are not politically abnormal are not pacifists; Nixon is a Republican as well.

The new information does not lead to a new proof that Nixon is a pacifist, but because it leads to a default proof that Nixon is *not* a pacifist, the truth value of $\mathrm{cl}(\phi)[\texttt{pacifist(Nixon)}]$ should be $*$ in this case. The expression (14) is not quite right; $\mathrm{cl}(\phi)(q)$ should also include information about the negation of q. We add to (14) an additional term dealing with $\neg q$ and elevate the result to the status of a definition:

Definition 3.4 *The* closure *of a truth assignment ϕ is the k-minimal truth assignment $cl(\phi)$ satisfying*

$$cl(\phi)(q) = \sum_{S \models q} u \bigvee \left[\bigwedge_{p \in S} cl(\phi)(p) \right] + \neg \sum_{S \models \neg q} u \bigvee \left[\bigwedge_{p \in S} cl(\phi)(p) \right] \qquad (23)$$

In our extended example, we can now compute the truth value assigned to the sentence $\mathrm{cl}(\phi)[\texttt{pacifist(Nixon)}]$ to be:

$$\begin{aligned} \mathrm{cl}(\phi)[\texttt{pacifist(Nixon)}] &= [u \vee (t \wedge t \wedge dt)] + [u \vee (t \wedge f)] + \neg[u \vee (t \wedge dt)] \\ &= [u \vee dt] + [u \vee f] + \neg[u \vee dt] \\ &= dt + u + df \\ &= * \end{aligned}$$

The flexibility of bilattice-based knowledge representation has been described in a variety of papers; the original [33] showed that the ideas generalize first-order logic, some instances of default reasoning [78], and assumption-based truth maintenance [10]. These results have been extended [34] to include first-order ATMSs and circumscription [66, 67]. Further applications to planning and reasoning about action will be discussed in the next section. The bilattice-based theorem prover MVL [37] is available by anonymous ftp from t.uoregon.edu [128.223.56.46]. Documentation is also available [41].

The work in this report is concerned not with general bilattices, but with *distributive* ones:

Definition 3.5 *A bilattice $(B, \wedge, \vee, \cdot, +, \neg)$ will be called* distributive *if each of the bilattice operations distributes with respect to the others, so that $x \wedge (y+z) = x \wedge y + x \wedge z$ and so on [33].*

For distributive bilattices, cl(ϕ) can be replaced with ϕ in the right hand side of the fixed-point equation (23):

Proposition 3.6 *Let ϕ be a truth assignment on a distributive bilattice. Then the closure of ϕ is given by:*

$$cl(\phi)(q) = \sum_{S \models q} u \bigvee \left[\bigwedge_{p \in S} \phi(p) \right] + \neg \sum_{S \models \neg q} u \bigvee \left[\bigwedge_{p \in S} \phi(p) \right] \tag{24}$$

where S ranges over those subsets of L that entail q or $\neg q$.[12]

It is this proposition (and its modal generalization 3.12) that underlie the algorithms we will discuss. The bilattices corresponding to first-order logic, ATMSs and first-order ATMSs, planning, and reasoning about action are all distributive. The setting is sufficiently powerful to describe nonmonotonic reasoning based on autoepistemic logic [70], PROLOG's negation-as-failure operator, and Kripke-style possible worlds semantics.

Modal operators What is a modal operator in this setting? We will simply take a modal operator to be any operator that is capable of manipulating the truth values of its sentential arguments:

Definition 3.7 *A* modal operator *is any n-ary function*

$$m : B^n \to B$$

from the bilattice of truth values to itself.

Given a modal operator m, we extend our declarative language to include sentences of the form $m(p_1, \ldots, p_n)$, where $p_i \in L$ for each i. Such sentences will be called modal.

As an example, we have already remarked that $\Box$ can be defined as the unary modal operator given by

$$\Box(x) = \begin{cases} t, & \text{if } x \geq_k t; \\ f, & \text{otherwise} \end{cases} \tag{25}$$

Thus $\Box p$ should be true if and only if p is known to be true. Note that $\Box(dt) = f$, since $t >_k dt$ and therefore $dt \not\geq_k t$. If p is labelled either false or unknown, then the truth value to be assigned to $\Box p$ should be false, since $\Box(f) = \Box(u) = f$ by virtue of (25).[13]

[12] Proofs appear in [42].

[13] The $\Box$ that we have defined here is not the same as the usual modal operator of necessity, although the two are closely related. In fact, an examination of earlier results [35] shows that if we take as our bilattice the set F^W of functions from a set W of possible worlds into the four-point bilattice F, then the semantics of our $\Box$ match the S5 semantics that are typically used. A complete discussion of the fact that the bilattice approach generalizes Kripke's can be found elsewhere [35].

Another example is the modal operator **1**, corresponding to the identity function. Even conjunction can be viewed modally; the truth value assigned to $p \wedge q$ inherits a component from the conjunction (in the bilattice sense) of the truth values assigned to p and to q. These two modal operators are rather different from $\Box$, however:

Definition 3.8 *A unary modal operator m will be called* monotonic *if it distributes with respect to $+$, so that $m(x+y) = m(x) + m(y)$ for all $x, y \in B$. A nonunary operator will be called monotonic if the result of fixing all but one of its arguments is monotonic.*

The modal operators **1** and $\wedge$ are monotonic on distributive bilattices; $\Box$ is not. That $\wedge$ is monotonic is simply a restatement of the distributivity axiom; to see that $\Box$ is not, note that

$$\Box(t+u) = \Box(t) = t$$

but

$$\Box(t) + \Box(u) = t + f = \bot$$

The distinction in Definition 3.8 is important because the semantics of monotonic and nonmonotonic modalities differ [35]. More specifically, if m is a nonmonotonic modality, the truth value of $m(p_i)$ can be computed from the truth values of the p_i's alone. This is not true for monotonic modalities: $p \wedge \neg p$ (which uses the monotonic modality $\wedge$) is false even if both p and $\neg p$ are unknown. Put slightly different, the truth value assigned to an expression involving a monotonic modality can depend in subtle ways on semantic interactions among the modal arguments.

The recursive descent description of modality that we discussed earlier depends fundamentally on our ability to evaluate a modal expression $m(p_i)$ by assigning values to the various p_i. Because only nonmonotonic modalities can be evaluated in this fashion, it is only nonmonotonic modal operators that mark interruption points.

A consequence of this is that the locations of interruption points are typically fixed by the semantic contents of the database in question. One cannot add new interruption points by scattering the modal operator **1** throughout the database; this has no semantic impact but also introduces no new interruptions because **1** is monotonic.

It also appears that the points marked by nonmonotonic modal operators are the same as those where it is natural to interrupt the inference process. But perhaps this is to be expected: A modal operator such as "It is necessarily the case that" is indeed a natural point at which to break reasoning. In a practical system, "It is necessarily the case that p" might mean that p appears explicitly in the database or, if more time is available for computation, that p is a deductive consequence of the material in the database.

The following is proved elsewhere [35]:

Proposition 3.9 *Any modal operator on a distributive bilattice can be written in terms of the modal operator of necessity $\Box$ and monotonic modalities.*

Given the argument in the previous paragraph justifying interruption at the modality $\Box$, Proposition 3.9 provides philosophical justification for interruption at any nonmonotonic modal operator.

It should now be no surprise that we will evaluate a modal expression $m(p_1, \ldots, p_n)$ either by invoking the prover recursively on the p_i to determine their truth values or by simply searching for the p_i in the database and using whatever values we find. The first approach is correct, but the second may provide a sensible approximation if computation time is limited.

Perlis has made a similar but informal suggestion [75]. He suggests that in databases with a modal operator of knowledge, it should be possible to determine whether or not $\mathtt{knows}(q)$ at any point in time, independent of the state of the analysis of q. This idea is essentially what we are trying to formalize here, working with a general description of modal operators as bilattice functions, and within a more precise formal setting.

What about closure in the presence of modality? Although (23) remains valid in this wider setting, the argument used to produce (24) by eliminating cl(ϕ) from the right hand side of (23) cannot be applied to monotonic modalities.

For nonmonotonic modalities, there is no problem because it is impossible to work "backwards" through a nonmonotonic modality K to conclude (for example) p from Kp. The standard example is Moore's and arises in autoepistemic logic [70]. Imagine a database containing the single statement, "If I had a brother, I'd know it." The obvious conclusion here is that since you don't know you have a brother, you don't have one, but there is an additional possibility where you do know that you have a brother, and it follows *from that* that you have a brother after all. This is wrong – you shouldn't be able to work your way backwards from a hypothetical belief to a conclusion about the state of the world generally. A groundedness condition is the technical mechanism that ensures this:

Definition 3.10 *A modal operator m will be called* grounded *if for any ϕ and sentence $m(p_i)$,*

$$cl(\phi)(m(p_i)) = m(cl(\phi)(p_i))$$

Proposition 3.11 *Nonmonotonic modal operators are grounded.*

The groundedness conditions associated with nonmonotonic modalities provide the semantic isolation needed by the inference procedures we are about to describe. We therefore assume that all modal operators are equipped with groundedness conditions, referring to such modal operators as *grounded* [35, Section 6]. This assumption might be valid because the operators are nonmonotonic, or because we are prepared to accept the limitations in expressiveness that the assumption entails. (For an example of such a limitation, see Section 3.2.)

In generalizing Proposition 3.6, we make use of the following notational convenience: If $S \subseteq L$ is a set of sentences, we denote by S_M the set of modal sentences in S and by S_0 the set of nonmodal sentences in S. We now have:

Proposition 3.12 *Let ϕ be a truth assignment and q a nonmodal sentence in our declarative language. Then*

$$cl(\phi)(q) = \sum_{S\models q} u \bigvee \left[\bigwedge_{p\in S_M} cl(\phi)(p) \wedge \bigwedge_{p\in S_0} \phi(p) \right] + \neg \sum_{S\models \neg q} u \bigvee \left[\bigwedge_{p\in S_M} cl(\phi)(p) \wedge \bigwedge_{p\in S_0} \phi(p) \right] \quad (26)$$

where S ranges over those subsets of L that entail q or $\neg q$.

Note that (26) is once again a fixed-point equation. However, the fact that cl(ϕ) is only applied to modal sentences in the right hand side of (26) means that the inherent circularity is restricted to situations where modal sentences are used in a proof of q. We will see shortly that this restriction has important practical implications.

At this point, suppose that we return to Fred and his ill-fated voyage; recall the PROLOG axiomatization appearing in Figure 2. Can we conclude dead(Fred)? We begin by presenting a method for translating PROLOG programs into our declarative language:

Definition 3.13 *Let s be the* PROLOG *statement*

```
h :- c1,...,cm,not(d1),...,not(dn).
```

By the declarative translation *of s, we will mean the sentence $D(s)$ given by*

$$c_1 \wedge \cdots \wedge c_m \wedge \neg\Box d_1 \wedge \cdots \wedge \neg\Box d_n \supset h$$

If P is a PROLOG *program, we will denote by $D(P)$ the conjunction of the declarative translations of each of the statements in P.*

The following is now an easy consequence of a variety of existing results [2, 35, 77, 104]:

Proposition 3.14 *Let P be a stratified* PROLOG *program. Then a positive atom* a *is a consequence of P if and only if $D(P) \models a$. A negative atom* not(a) *is a consequence of P if and only if $D(P) \models \neg\Box a$.*

In this particular example, the declarative translation of our PROLOG program is the following, where all of the variables are universally quantified:

$$\begin{aligned}
\texttt{falling-onto}(x,y) \wedge \neg\Box\texttt{good-parachute}(x) \wedge \neg\Box\texttt{soft}(y) &\supset \texttt{dead}(x) && (27)\\
\texttt{wearing-parachute}(x,p) \wedge \neg\Box\texttt{broken}(p) &\supset \texttt{good-parachute}(x) && (28)\\
\texttt{haystack}(y,h) \wedge \neg\Box\texttt{ouch}(h) \wedge \neg\Box\texttt{missed}(h) &\supset \texttt{soft}(y) && (29)\\
\texttt{pitchfork}(h,p) \wedge \neg\Box\texttt{missed}(p) &\supset \texttt{ouch}(h) && (30)
\end{aligned}$$

$$\begin{gathered}\texttt{falling-onto(Fred,field)}\\ \texttt{wearing-parachute(Fred,parachute)}\\ \texttt{broken(parachute)}\\ \texttt{haystack(field,haystack)}\\ \texttt{pitchfork(haystack,pitchfork)}\\ \texttt{missed(pitchfork)}\\ \texttt{missed(haystack)}\end{gathered}$$

Our initial truth assignment ϕ labels all of the above sentences with t.

Since missed(haystack) is labelled as true by ϕ, the truth assignment $\mathrm{cl}(\phi)$ will also need to label missed(haystack) as true; $\Box$missed(haystack) will therefore be true as well. This means that $\neg\Box$missed(haystack) will be labelled false and there is no reason for us to label soft(field) as anything other than unknown because (29) cannot be applied.

With soft(field) labelled unknown, it follows that $\Box$soft(field) will be labelled false, so that $\neg\Box$soft(field) will be labelled t. We can conclude in a similar way that $\neg\Box$good-parachute(Fred) is also true, and can now use (27) to conclude that Fred does indeed die.[14]

We were able to assign a truth value first to the sentences good-parachute(Fred) and to ouch(haystack), then to soft(field), and finally to dead(Fred) because the database with which we are working is of a form that supports this sort of hierarchical reasoning. To formalize this, we make the following definition:

Definition 3.15 *Let ϕ be a truth assignment. By a* syntactic stratification *of ϕ we will mean a mapping s from the set of atoms into the nonnegative integers such that for any pair of atoms a_1 and a_2 appearing in a single sentence p with $\phi(p) \neq u$:*

1. *If both a_1 and a_2 appear outside the scope of all modal operators in p, then $s(a_1) = s(a_2)$.*

2. *If a_1 appears under the scope of a modal operator and a_2 does not, then $s(a_1) < s(a_2)$.*

If both a_1 and a_2 appear only under the scope of modal operators, no restriction is entailed. Furthermore, if a_1 or a_2 is an instance of an atom appearing in p, the same conditions apply.

A truth assignment ϕ will be called syntactically stratified *if there is some function s that is a syntactic stratification of ϕ.*

In the example we have been considering, Definition 3.15 applied to (27) – (30) gives us:

$$\begin{gathered}s[\texttt{falling-onto}(x,y)] = s[\texttt{dead}(x)] > s[\texttt{good-parachute}(x)], s[\texttt{soft}(y)]\\ s[\texttt{wearing-parachute}(x,p)] = s[\texttt{good-parachute}(x)] > s[\texttt{broken}(p)]\\ s[\texttt{haystack}(y,h)] = s[\texttt{soft}(y)] > s[\texttt{ouch}(h)], s[\texttt{missed}(h)]\\ s[\texttt{pitchfork}(h,p)] = s[\texttt{ouch}(h)] > s[\texttt{missed}(p)]\end{gathered}$$

[14] There are other possibilities in which, for example, soft(field) is labelled t and dead(Fred) remains unknown. These extensions are not grounded, however [35].

for any x, y, h and p. We can therefore take

$$\begin{aligned}
s[\texttt{missed}(z)] = s[\texttt{broken}(p)] &= 0\\
s[\texttt{pitchfork}(h,p)] = s[\texttt{ouch}(h)] &= 1\\
s[\texttt{wearing-parachute}(x,p)] = s[\texttt{good-parachute}(x)] &= 1\\
s[\texttt{haystack}(y,h)] = s[\texttt{soft}(y)] &= 2\\
s[\texttt{falling-onto}(x,y)] = s[\texttt{dead}(x)] &= 3
\end{aligned}$$

This syntactic stratification matches Figure 3; the value assigned to any sentence is the distance of that sentence from the fringe of the modal tree. The figure only shows sentences appearing under the scope of modal operators; we have (for example)

$$s[\texttt{falling-onto}(x,y)] = s[\texttt{dead}(x)]$$

because they have identical status in (27).

We first determine truth values for sentences with $s = 0$, deciding that Fred misses the haystack and the pitchfork and that his parachute is broken. Next is $s = 1$: There is a pitchfork in the haystack but Fred misses it so it won't hurt to land there; Fred is wearing a parachute but it's not going to do him any good. At $s = 2$ we conclude that there is a haystack in the field but that it won't help to cushion Fred's fall. Finally, at $s = 3$ we realize that Fred is going to die because he's falling onto the field (and will miss the haystack).

The point of a syntactic stratification is that it allows us to compute the closure of a truth assignment incrementally, ignoring at step i any sentence involving a literal l with $s(l) > i$. We also have the following:

Proposition 3.16 *Let P be a* PROLOG *program. Then if $D(P)$ is syntactically stratified, P is stratified.*

Of course, this result is in the wrong direction, showing the PROLOG community's definition of stratification to be more general than ours. We generalize our definition of stratification substantially in Section 3.1.3.

Before moving on, let us summarize the results we have reviewed thus far:

1. It is possible to assign modal operators a truth-functional semantics. Furthermore, it can be shown [35] that in a bilattice setting, this semantics is a generalization of the standard Kripke semantics [62].

2. By determining the truth value of sentences before determining the truth values of the modal expressions to which they contribute, it is sometimes possible to stratify a declarative database in a way that allows incremental computation. Note that this stratification does not in and of itself imply the existence of an anytime algorithm; this issue is the contribution of the research on which we are reporting, and we turn to it now.

3.1.2 Anytime reasoning

In this section, we use Proposition 3.12 to develop an anytime approach to inference. Here is (26) again:

$$\mathrm{cl}(\phi)(q) = \sum_{S \models q} u \bigvee \left[\bigwedge_{p \in S_M} \mathrm{cl}(\phi)(p) \wedge \bigwedge_{p \in S_0} \phi(p) \right] + \neg \sum_{S \models \neg q} u \bigvee \left[\bigwedge_{p \in S_M} \mathrm{cl}(\phi)(p) \wedge \bigwedge_{p \in S_0} \phi(p) \right]$$

As we remarked earlier, $\mathrm{cl}(\phi)$ is only applied to modal sentences in the right hand side of the above expression; if ϕ is stratified, this enables us to reduce the problem of assigning a truth value to q to the problem of assigning truth values to sentences on lower stratification levels than q.

To make practical use of this idea, suppose that we are trying to determine the value assigned to q by $\mathrm{cl}(\phi)$. Using (26), we reduce the problem to that of assigning truth values to the sentences $p_1, \ldots, p_n$ appearing under the scope of modal operators used in proofs of q. Each p_i can now be analyzed similarly. If the database is syntactically stratified, this process is guaranteed to terminate and we can trickle the results back to determine the values assigned to each of the intermediate sentences and, eventually, to the original query q. We construct Figure 3 from the root downwards and then trickle the answer back to the original query.

The following procedure implements this. We first find possible proofs of the user's query q or of other sentences on which q depends, and then compute truth values that will be assigned by $\mathrm{cl}(\phi)$ to these sentences.

Procedure 3.17 Given a truth assignment ϕ, to respond to a nonmodal query q:

```
$J :=$ find-proofs($\phi, q$)          ($J$ contains justification information)
$\phi^J :=$ update($\phi, J$)          ($\phi^J$ is a version of $\phi$ that takes this justification
                                         information into account)
if $\phi^J =$ failure then return failure else return $\phi^J(q)$
```

To find the proofs that bear on the query q, we maintain three lists. The first list U is a list of unexpanded nodes in the modal tree. This list thus contains propositions that still need to be considered by the system because they appear in modal expressions used to prove the original query q, its negation, or other sentences generated by the proof process. Expanded nodes correspond to propositions that have already been considered are retained in the second list E. The final list J associates, to each of these propositions, information about the sets S that entail it. Note also that (26) implies that the only information we need to retain about a particular set S is a list of the elements of S_M together with the truth value $\bigwedge_{p \in S_0} \phi(p)$.

Procedure 3.18 To compute `find-proofs`(ϕ, q), justification information relevant to q given the truth assignment ϕ:

```
$J := \emptyset$
$U := \{q, \neg q\}$                    ($U$ is a list of propositions still to be investigated)
$E := \emptyset$                        ($E$ is a list of sentences that have been considered)
repeat until $U = \emptyset$:
        $p :=$ an element of $U$
        $S :=$ the sentences used in a new proof of $p$
        if such an $S$ can be found
          then $J := J \cup \{\langle p, \bigwedge_{w \in S_0} \phi(w), S_M \rangle\}$
                 $U := U \cup [\{$sentences appearing inside elements of $S_M$ or
                                their negations$\}] - E$
          else remove $p$ from $U$ and add it to $E$
        end if
end repeat
return $J$
```

Having constructed the tree, we need to work through the entries in J and assign updated truth values to every sentence that appears there in accordance with (26); in the following procedure, R is a list of the nodes that have not yet been assigned values. At each step, we find a sentence all of whose modal dependencies (i.e., children) have been evaluated, assign it an appropriate truth value using (26), and then remove it from the list R. If we ever fail to find such a sentence, there must be a loop in the modal dependencies in the original database, and the procedure fails. (This can only happen for nonstratified databases.) The procedure returns a modified version of the original truth assignment ϕ.

Procedure 3.19 To compute update(ϕ, J), the result of updating the truth assignment ϕ to incorporate information from the justifications in J:

```
$R := \{p | p$ or $\neg p$ appears as the leading element of a triple in $J\}$
                (R contains sentences that need to have their truth values computed)
$\phi^J := \phi$
**repeat** until $R = \emptyset$:
    $C := \{p \in R |$for all $\langle p, v, S_M\rangle$ or $\langle \neg p, v, S_M\rangle$ in $J$, $S_M \cap R = \emptyset\}$
    **if** $C = \emptyset$ **then return** failure
    $p :=$ any element of $C$
    $\phi^J(p) := \sum_{\langle p,v,S_M\rangle \in J} u \bigvee \left[\bigwedge_{m(p_i)\in S_M} m(\phi^J(p_i)) \wedge v\right] +$
              $\neg \sum_{\langle \neg p,v,S_M\rangle \in J} u \bigvee \left[\bigwedge_{m(p_i)\in S_M} m(\phi^J(p_i)) \wedge v\right]$
    $\phi^J(\neg p) := \neg\phi^J(p)$
    $R := R - \{p, \neg p\}$
**end repeat**
**return** $\phi^J$
```

As an example, suppose that we use the procedure to respond to our usual question about Fred. In order to make things a bit more compact, we rewrite our database as:[15]

$$\begin{gathered} j \wedge \neg\Box g \wedge \neg\Box s \supset d \\ w \wedge \neg\Box b \supset g \\ h \wedge \neg\Box o \wedge \neg\Box m_h \supset s \\ p \wedge \neg\Box m_p \supset o \\ j \wedge w \wedge b \wedge h \wedge p \wedge m_p \wedge m_h \end{gathered} \tag{31}$$

The first part of the analysis is summarized in the following table:

	U	E	J
1	d		
2	g, s	d	$\langle d, t, \{\neg\Box g, \neg\Box s\}\rangle$
3	b, s	g	$\langle g, t, \neg\Box b\rangle$
4	s	b	$\langle b, t, \emptyset\rangle$
5	o, m_h	s	$\langle s, t, \{\neg\Box o, \neg\Box m_h\}\rangle$
6	m_p, m_h	o	$\langle o, t, \neg\Box m_p\rangle$
7	m_h	m_p	$\langle m_p, t, \emptyset\rangle$
8	$\emptyset$	m_h	$\langle m_h, t, \emptyset\rangle$

(32)

We show only the atoms contained in U or E; as an example, U is initially set to $\{d, \neg d\}$ but we show only d. Since the set E of analyzed sentences and the set J of justifications grow

[15] The mnemonics are as follows: j is jump out of the plane, g is good parachute, s is a soft field, d is die, w is wear a parachute, b is a broken parachute, h is a haystack, o is ouch, m_x is miss x and p is a pitchfork.

monotonically as the procedure is executed, each entry above includes only that element, if any, that is newly added to E or to J. The entries in J themselves are of the form

$$\langle w, x, \{p_1, \ldots, p_n\}\rangle$$

where w can be proven from a set S with $x = \wedge_{w \in S_0} \phi(w)$ and $S_M = \{p_1, \ldots, p_n\}$.

In detail, what we have done is the following:

1. Initially, E and J are set to $\emptyset$ and d (the query) is added to U. The negation of d is also added, but we don't show it.

2. The only proof of d or $\neg d$ from other information in the database is that d is a consequence of j (which itself is a consequence of (31) and so has truth value t), $\neg\Box g$ and $\neg\Box s$. We add g and s to U, move d from U to E, and add the indicated triple to J. The inference that we have drawn corresponds to the commonsense conclusion that people falling from airplanes typically die.

3. Searching for proofs of g or $\neg g$, we see that g follows from w and $\neg\Box b$, so b is added to U and a second entry is added to J. Fred's parachute will save him if it isn't broken.

4. Turning our attention to b, we see that b is a consequence of (31), so its truth value is t. This leads to the addition of an entry $\langle b, t, \emptyset\rangle$ to J; the third element of this entry is empty because no modal assumptions were used in the proof.

5. The remainder of (32) considers s (is the field soft?), o (is there a pitchfork around?), m_p (did Fred miss the pitchfork?) and m_h (did he miss the haystack?) in that order. U is now empty, and this phase of the analysis is complete.

A variety of choices were made as the procedure was executed. For example, we decided to consider g before s. The reason that we selected this particular order was to match the order of information in the original story, although the final answer is independent of the choices made (see Proposition 3.20). We will see shortly that the anytime version of this procedure may well be sensitive to the order in which proofs are discovered.

We now move to the second phase of the procedure – extracting the answer from the above table. This proceeds as follows:

	R	q	$\phi^J(q)$
1	d, g, b, s, o, m_p, m_h		
2	d, g, b, s, o, m_p	m_h	t
3	d, g, b, s, o	m_p	t
4	d, g, b, s	o	u
5	d, g, b	s	u
6	d, g	b	t
7	d	g	u
8	$\emptyset$	d	t

(33)

1. Initially, we set $\phi^J = \phi$.

2. Since m_h does not depend on any other modal sentences, we can compute the truth value assigned to m_h to be t. Fred misses the haystack. The sentence m_p is true similarly.

3. Continuing, o (ouch) depends on $\neg\Box m_p$ in order to be true. Since Fred does miss the pitchfork, o remains unknown.

4. s (the field is soft) will be true if the field contains a safe haystack (which it does, since $\neg\Box o$ is true because o is unknown) and if Fred hits the haystack (which he doesn't since $\neg\Box m_h$ is false because m_h is true). Since Fred misses the haystack, s remains unknown.

5. As for m_h and m_p, b is true because it has no other modal dependencies. g is unknown for reasons similar to s.

6. Finally, we evaluate the truth value to be assigned to d. Since $\phi^J(s) = \phi^J(g) = u$ (we couldn't prove either that Fred would land in a haystack or that he was wearing a working parachute), we assign f to both $\Box s$ and $\Box g$, and therefore t to both $\neg\Box s$ and $\neg\Box g$. This leads us to assign t to d as well, and we return t as a response to the original query. Fred dies.

Proposition 3.20 *If Procedure 3.17 terminates successfully, it will return* $cl(\phi)(q)$.

Proposition 3.21 *Procedure 3.17 will terminate successfully provided that:*

1. *All calls to the underlying first-order theorem prover themselves terminate,*

2. *Only a finite number of proofs are found for any particular sentence in the database, and*

3. ϕ *is syntactically stratified.*

A consequence of Propositions 3.20 and 3.21 is that we have indeed managed to capture the meanings of PROLOG programs in our setting. And it might seem that we have done nothing more; Procedure 3.17 seems to be going to great lengths to do what PROLOG interpreters do already. In some sense this is true, although an advantage of working in our setting is that our results apply to modal operators generally as opposed to only the negation-as-failure operator used in PROLOG programs.

Far more interesting, however, is the following. Suppose that we change the termination criteria for the loop at the beginning of Procedure 3.18 to be:

repeat until either $U = \emptyset$ **or no time remains for additional analysis** ...[16]

[16] The "time remaining for additional analysis" need not include the time required by the second phase of the algorithm; this phase involves simply passing truth values up the tree and is effectively instantaneous. Alternatively, the two phases of the algorithm can be interleaved, so that the current estimate of $cl(\phi)(q)$ is available at all times.

As an example, suppose that we decide to terminate the analysis about Fred after step 7 of the inferential phase, so that we have not yet proved that Fred will miss the haystack. Now (33) becomes

	R	q	$\phi^J(q)$
1	d, g, b, s, o, m_p		
3	d, g, b, s, o	m_p	t
4	d, g, b, s	o	t
5	d, g, b	s	t
6	d, g	b	t
7	d	g	u
8	$\emptyset$	d	u

Since J doesn't contain any entries involving m_h, we assign m_h the truth value unknown. (Note that m_h does not appear *explicitly* in the database; it is only a consequence of (31).) We now conclude that s is true – since Fred misses the pitchfork, the field is a safe place to land. This causes us to label $\neg\Box s$ as false, and we no longer draw the conclusion that Fred dies.

Here are the conclusions that would be drawn if we were to terminate our analysis after each inference step:

	U	E	J	$\phi^J(d)$
1	d			u
2	g, s	d	$\langle d, t, \{\neg\Box g, \neg\Box s\}\rangle$	t
3	s, b	g	$\langle g, t, \neg\Box b\rangle$	u
4	s	b	$\langle b, t, \emptyset\rangle$	t
5	o, m_h	s	$\langle s, t, \{\neg\Box o, \neg\Box m_h\}\rangle$	u
6	m_p, m_h	o	$\langle o, t, \neg\Box m_p\rangle$	t
7	m_h	m_p	$\langle m_p, t, \emptyset\rangle$	u
8	$\emptyset$	m_h	$\langle m_h, t, \emptyset\rangle$	t

(34)

After the second step, we realize that people falling out of airplanes typically die. The third step introduces the parachute; since b does not appear explicitly in the database (it is only a consequence of (31)), we label $\Box b$ as false, $\neg\Box b$ as true, g as true, and therefore d as unknown. At step 4, we realize that b is true after all, and conclude that Fred dies.

At step 5, we realize that the field contains a haystack; since m_h is not explicitly in the database, Fred's outlook once again improves. We observe the presence of the pitchfork in step 6 and realize that Fred will miss it in step 7. Finally, we realize in step 8 that Fred will miss the haystack and be killed after all. All of this is in keeping with the tenor of the original story.

Note that although we allow the inference procedure to be interrupted between invocations of the first-order prover, the first-order prover itself is not interrupted. The implementation works by modifying the database lookup mechanism so that it succeeds on any modal expression and then invoking the prover on p and allowing it to run until a proof is found.

Corollary 3.22 *Given unlimited computational resources, the value returned by the modified Procedure 3.17 in response to a query q is $cl(\phi)(q)$.*

It is now reasonable to think of the modified procedure as anytime, although some slight subtleties are involved:

1. The modified procedure does not have an answer available at any time; we only compute the value of $\phi^J(q)$ when we have decided to terminate the first phase of the analysis. This is simple to change, however: It is easy to update ϕ^J whenever a new triple is added to J and thereby to ensure that an approximate answer is available at all times.

2. That the answer produced converges is guaranteed by Corollary 3.22. That the convergence is in some sense uniform is a bit harder to argue, since our conclusions change drastically from one step of the analysis to the next. Indeed, it is quite possible for the procedure to return the wrong answer at every point until the analysis is complete.

   The answers returned by our procedure are of uniformly improving quality in the sense that each considers more information than the previous one, since allowing more time for analysis monotonically extends the set J used by the procedure to cache intermediate conclusions. At any point in time, we make use of all information that has been determined to be relevant to the query.[17] We are not the only ones to view this sort of behavior as anytime; Drummond [17] makes similar remarks about a universal-planning algorithm that gradually considers more hypothetical situations in which an agent might find itself.

   In a fully declarative setting, this uniformity of information (but not of answer) is probably the best one can do. Agents embedded in real-time environments have no choice but to make mistakes; they will fail to consider some aspect of a problem that turns out to be relevant and draw conclusions that are simply wrong. Forming a conclusion, reversing it, and then reversing it again is something that we have all done; there is no reason to expect artificial systems to perform otherwise.

3. Although Corollary 3.22 guarantees that the final answer returned is independent of the syntactic structure of the database, the intermediate answers may well be sensitive to these syntactic features. As an example, if we had replaced (31) with seven individual axioms (one for each conjunct), we would have already been able to assign b the truth value t in step 3 of (34), thereby continuing to conclude that Fred dies. The intermediate answers are also sensitive to the order in which various subgoals are considered; had we noticed earlier that Fred missed the haystack, the subargument centered on the pitchfork could have been ignored.

4. Finally, although the eventual convergence of the procedure is guaranteed, we have presented no mechanism for estimating the computational resources needed to provide this convergence. It seems unlikely that there will be a sharp theoretical result to be

[17] Becky Thomas provided us with this argument.

had here, since such a result would depend on an ability to bound the number of proofs that will be found for any particular sentence in U.

3.1.3 Semantic stratification

Consider our example once again. When investigating the sentence good-parachute(Fred), the details of Procedure 3.17 require us to look not only for proofs that Fred has a working parachute, but also proofs that he *doesn't* have one. To see why, suppose that (27) were replaced with

$$\texttt{falling-onto}(x,y) \wedge \neg M\texttt{good-parachute}(x) \wedge \neg M\texttt{soft}(y) \supset \texttt{dead}(x) \tag{35}$$

where the modal operator M ("maybe") indicates that x *might* have a working parachute. The above axiom says that people falling out of airplanes die if they are known *not* to have working parachutes and if they are known to be falling toward hard objects. This axiom would be suitable if people typically wore parachutes and the ground were made of Jello.

Now showing that good-parachute(Fred) has truth value f instead of u will cause the truth value of Mgood-parachute(Fred) to change from t to f, and this will impact our ability to use (35).

With the original axiom (27), however, there is no way that an explicit proof that Fred has no working parachute can affect our conclusion that he dies. This is because good-parachute(Fred) affects our conclusions only via the modal sentence

$$\Box\texttt{good-parachute(Fred)} \tag{36}$$

and if the truth value of good-parachute(Fred) changes from unknown to false there will be no impact on (36), since □good-parachute(Fred) is false in either case. The next step in our development of Procedure 3.17 is to modify it to cater to this possibility.

Suppose that during our execution of Procedure 3.17, we have constructed a set J of justification triples that update will use to respond to the original query q. Given an arbitrary set J of justifications and truth assignment ϕ, we continue to denote by ϕ^J the truth assignment update(ϕ, J) computed by Procedure 3.19.

Now suppose that we have a fixed sentence p; can a proof of p be relevant to the query q? If J is the set of justifications accumulated so far, then p will clearly be relevant to q if

$$\psi^J(q) \neq \phi^J(q) \tag{37}$$

where ψ agrees with ϕ except that $\psi(p) = t$, so that ψ is true at p. In informal terms, ψ and ψ^J correspond to what we would know if the attempt to prove p succeeded; ϕ and ϕ^J reflect what we know currently.

In fact, p can affect the truth value eventually assigned to q any time that p's truth value changes from its current value $\phi(p)$. To see why, suppose that we are working in the default bilattice and have a modal operator m with $m(u) = m(t) = u$ but $m(dt) = t$. Now if we have a sentence like

$$m(p) \supset q \tag{38}$$

it will certainly be important to consider proofs of p, since they might lead us to conclude that p is true by default and therefore that q is true. This leads us to the following definition:

Definition 3.23 *Given truth assignments ψ and ϕ and a set of sentences S, we will say that ψ* (positively) extends ϕ at S *if ψ is a (positive) extension of ϕ for which $\psi(x) = \phi(x)$ for all $x \notin S$.*

The intuition behind (37) corresponds to the observation that p will be relevant to q if there is some ψ that positively extends ϕ at p and such that

$$\psi^J(q) \neq \phi^J(q) \tag{39}$$

Overall, things aren't quite this simple. Consider (35); is

$$\neg\texttt{good-parachute(Fred)}$$

relevant to `dead(Fred)`? Not by virtue of (39), since changing the truth value of

$$\neg\texttt{good-parachute(Fred)}$$

from u to t isn't enough to change our belief that Fred survives the fall; we *also* have to know that the truth value of $\neg$`soft(field)` changes from u to t.

Although we have to make assumptions about other sentences when determining the relevance of

$$\neg\texttt{good-parachute(Fred)}$$

to our original query, the sentences about which these assumptions are made are always elements of U. What we are doing is imagining that we in fact will succeed in our eventual search for a proof of some other sentence $w \in U$.

Definition 3.24 *Given a truth assignment ϕ, a query q, a set of sentences U and a set of justification triples J, we will say that p is* relevant to q *if there is some χ that positively extends ϕ at U and some ψ that positively extends χ at p such that $\psi^J(q) \neq \chi^J(q)$.*

In other words, p is relevant to q if, assuming that we learn enough to replace ϕ with χ, a change in p's truth value will affect the truth value to be assigned to q.

We are now in a position to rewrite Procedure 3.17. In the subroutine that computes the justification information, we require that the proposition being considered be relevant to the original query. We also include a trivial modification to deal with modal queries:

Procedure 3.25 Given a truth assignment ϕ, to respond to a query q:

```
$J$ := find-relevant-proofs($\phi, q$)
$\phi^J$ := update($\phi, J$)
if $\phi^J$ = failure then return failure else return $\phi^J(q)$
```

Procedure 3.26 To compute `find-relevant-proofs`(ϕ, q), justification information relevant to U given the truth assignment ϕ:

```
$J := \emptyset$
if $q$ is modal with $q = m(q_i)$, then $U := \{q_i, \neg q_i\}$ else $U := \{q, \neg q\}$            [1]
$E := \emptyset$
repeat until $U = \emptyset$ or no time remains for additional analysis:                        [2]
        $p :=$ an element of $U$ that is relevant to $q$                                           [3]
        if no such $p$ can be found then $U := \emptyset$
          else $S :=$ the sentences used in a new proof of $p$
              if such an $S$ can be found
                then $J := J \cup \{\langle p, \bigwedge_{w \in S_0} \phi(w), S_M \rangle\}$
                      $U := U \cup [\{$sentences appearing inside elements of $S_M$ or
                                    their negations$\}] - E$
                else remove $p$ from $U$ and add it to $E$
              end if
          end else
        end if
end repeat
return $J$
```

The changes are marked in the right margin:

1. If the query q is modal, we initially set U to contain the sentential arguments of the modal operator involved.

2. We terminate the execution of the algorithm if no time remains for further work.

3. We only analyze sentences that are relevant to the original query.

An implementation of this procedure needs some effective way to decide whether a particular sentence p or its negation is relevant to the original query q. This is achieved in practice by assuming that we can always take $\psi(p) = t$ in Definition 3.24, so that there is no effect such as that in the example surrounding (38). All naturally occurring modalities m appear to either preserve the t ordering or to invert it, so that if $x \geq_t y$ then uniformly either $m(x) \geq_t m(y)$ or $m(x) \leq_t m(y)$. This condition is enough to support the above assumption.[18]

We can now determine if p is relevant to q by considering p in combination with every subset of the set U of sentences currently under consideration. The structure of the modal dependencies can be used to prune substantially the number of possibilities considered, and

[18] As examples, $\Box$ preserves t and $\neg$ inverts it. Exceptions such as $\Box + \neg\Box$ seem to have no naturally occurring interpretations.

many intermediate computations can be cached by keeping ϕ^J updated at all times and only computing differences between this truth assignment and a hypothetical one.

Corollary 3.27 *Given unlimited computational resources, the value returned by Procedure 3.25 in response to a query q is $cl(\phi)(q)$.*

The modification that produced Procedure 3.25 from Procedure 3.17 is important for a variety of reasons. The first is the one that led to the change in the first place; as an example, suppose that we include in our original axiomatization the sentence

$$\texttt{hard-to-prove} \supset \neg\texttt{good-parachute(Fred)}$$

so that if we can derive `hard-to-prove`, we will know for certain that Fred has no good parachute.

Procedure 3.17 would now have us attempt to derive `hard-to-prove`, since doing so will affect the truth value assigned to $\neg g$ and therefore potentially impact modal expressions depending on g. However, since the only modal expression involved is $\Box g$ and this expression is unaffected by a change in truth value of g from u to f, we need not consider `hard-to-prove` after all. Procedure 3.25 acts in accordance with this observation.

There are also examples where Procedure 3.25 succeeds but Procedure 3.17 doesn't. Here is one:

```
landlubber(X) :- animal(X), not(fly(X)).
fly(X) :- bird(X), not(penguin(X)).
penguin(X) :- bird(X), tuxedo(X).
animal(X) :- bird(X).
tuxedo(opus).
bird(opus).
```

Animals are typically landlubbers; birds are typically flyers although penguins are an exception. Birds in tuxedos are penguins, and birds are animals. Opus is a bird in a tuxedo. Is he a landlubber?

The above axioms do not meet our narrow definition of stratification. The second axiom requires that we have

$$s[\texttt{bird}(x)] > s[\texttt{penguin}(x)]$$

while the third forces

$$s[\texttt{bird}(x)] = s[\texttt{penguin}(x)]$$

a contradiction. At this point, however, Procedure 3.17 still terminates successfully. Here is a trace similar to (32), where we have written l for `landlubber(Opus)`, *fly* for `fly(Opus)` and p for `penguin(Opus)`:

	U	E	J
1	l		
2	fly	l	$\langle l, t, \neg\Box fly\rangle$
3	p	fly	$\langle fly, t, \neg\Box p\rangle$
4	$\emptyset$	p	$\langle p, t, \emptyset\rangle$

Invoking the prover on landlubber(Opus), we need to prove both animal(Opus) and $\neg\Box$fly(Opus). The last sentence is modal and will therefore be passed to subsequent steps of the algorithm; we can derive animal(Opus) from the fact that Opus is a bird. In other words, if $\neg\Box$fly(Opus) is true, Opus will be a landlubber. There is no other potential proof that Opus is a landlubber from information in the database.

To decide whether or not Opus can fly, we note that since he is a bird, he can fly unless he is known to be a penguin. We conclude that he is a penguin because he is a bird in a tuxedo.

We now modify our axiomatization to include the fact that penguins can't fly:

$$\texttt{penguin}(x) \supset \neg\texttt{fly}(x)$$

This seems innocuous enough, but leads to a proof that Opus is not a penguin. Since penguins can't fly, anything that can fly isn't a penguin. Since Opus is a bird, he can fly unless he is known to be a penguin. This leads to the conclusion that

$$\neg\Box\texttt{penguin(Opus)} \supset \neg\texttt{penguin(Opus)} \tag{40}$$

This makes sense: Since penguins are abnormal birds, we assume if possible that Opus is not a penguin.

Unfortunately, (40) causes us to add the following triple to J:

$$\langle \neg p, t, \neg\Box p\rangle \tag{41}$$

where p denotes the sentence penguin(Opus). Since this triple makes it impossible to compute the truth value of p, Procedure 3.17 is doomed to fail.

Not so for Procedure 3.25. Since the effect of (41) is to change the truth value of p either from u to f (unknown to false) or from t to $\perp$ (true to a contradiction) and the value assigned to $\Box p$ is unchanged in either case (f in the first case and t in the second), $\neg p$ will never be relevant to l and we are able to conclude that Opus is a landlubber as previously.

The point here is that Procedure 3.25 can succeed for databases that are not syntactically stratified or for which Procedure 3.17 fails. Coupling this observation with Corollary 3.27, which tells us that the procedure is actually computing a semantic object (the truth value to be assigned to q in the closure), we are led to the following definition:

Definition 3.28 *A truth assignment ϕ will be called* semantically stratified *if Procedure 3.25 never fails when applied to ϕ.*

It might seem that this definition is too procedural to be of much use, but this is not the case; we will see in the next section, for example, that any stratified logic program is semantically stratified. We immediately have:

Proposition 3.29 *Any syntactically stratified truth assignment is semantically stratified.*

What we have shown thus far is that it is possible to view grounded modal operators as semantic markers for points at which a declarative inference procedure can be suspended, and that for semantically stratified databases, an anytime inference procedure can be built around this idea. The point that we have *not* addressed is to what extent interesting databases are semantically stratified.

3.1.4 Subsumption of existing work

We have presented an algorithm for anytime reasoning, shown that it terminates for a restricted class of declarative databases and shown that if it does terminate, it will return the answer supported by the multivalued semantics of modality.

Results such as these are quite parochial; we need to relate our techniques to existing work. We do that now, showing that our view of modality subsumes a variety of existing results from logic programming and knowledge representation.

Logic programs The notion of stratification is not unique to us, of course; it originated in the logic programming community. Here is a stratified logic program:

```
a :- not(b).
a :- b.
```

From a logic programming point of view, the second axiom is "about" a and not b, so that we can stratify the program by determining the truth value of b before that of a.

From our point of view, however, the declarative translation of the second axiom is $b \supset a$, and is just as likely to allow us to conclude $\neg b$ from $\neg a$ as it is to allow us to conclude a from b. However, since the axiom $b \supset a$ can never be used to prove that b is *true*, we can still conclude that $\Box b$ is false and therefore that a follows in this example. The declarative translation of the above program fails to meet the requirements of Definition 3.15, but does satisfy Definition 3.28.

Here is a somewhat more formal way to look at it. The definition of stratification used by the logic programming community is the following [2]:

Definition 3.30 *Let P be a logic program. By an* LP-stratification *of P we will mean a mapping s from the set of atoms in P into the positive integers such that for every sentence*

```
a :- b1, ... bm, not(c1), ... not(cn).
```

we have $s(a) \geq s(b_i)$ for each i and $s(a) > s(c_j)$ for each j.

A logic program P will be called LP-stratified *if there is some function s that is an LP-stratification of P.*

Our definition of stratification requires $s(a) = s(b_i)$ for each b_i, since the declarative translation of (for example)

```
a :- b, not(c).
```

allows each of a and b to influence the other. Definition 3.30 only requires $s(a) \geq s(b_i)$ because it is realized that PROLOG cannot use information about a to draw conclusions about any of the b_i's. We can extend Proposition 3.16 to show that our definition is more general than that of the logic programming community:

Proposition 3.31 *If P is an LP-stratified logic program, then $D(P)$, the declarative translation of P, is a semantically stratified truth assignment.*

The converse of this result does not hold. Imagine, for example, that there is a strange genetic trait that skips generations. We might axiomatize this as:

$$\texttt{strange(father(X)) :- not(strange(X)).} \tag{42}$$

saying that if x lacks this genetic characteristic, x's father will have it. This axiom resembles an example of Van Gelder's [104].

An axiomatization including (42) can never be stratified because the same predicate occurs on both sides of the implication. In practice, however, investigation of

$$\texttt{strange}(\texttt{father}^k(x))$$

will lead us to consider $\texttt{strange}(\texttt{father}^{k-1}(x))$ so that no loop occurs. Members of the logic programming community have covered this sort of example by replacing (42) with all of its ground instances and modifying Definition 3.30 to deal with ground instances only [77, 104]. The resulting definition is that of *local* stratification, and a program including (42) can indeed be locally stratified. Such programs are typically semantically stratified for similar reasons.

Here is a PROLOG fragment that is not even locally stratified:

```
a :- not(b).
b :- not(a).
c :- not(d).
```

This fragment admits two minimal models. In one, a is true but b is not; in the other, b is true but a is not. Although this fragment is not semantically stratified either, Procedure 3.25 continues to terminate successfully when applied to c, since an investigation of c will never encounter the a-b loop.

Procedure 3.25 is not a panacea, however. Consider:

```
a :- not(b).
b :- not(a).
c :- a.
c :- b.
```

On an intuitive level, the query c should perhaps succeed, since c is a consequence of $a \vee b$ and this latter sentence holds in all minimal models of the above program. Although the formal semantics support this conclusion [35], Procedure 3.25 fails to return an answer.

Negative subgoals We have focussed thus far on applying our ideas to a specific modal operator, the □ appearing in PROLOG programs. But the ideas can be applied to other modalities as well, allowing us to generalize work from a variety of other areas. One specific example involves the following program fragment:

```
g(Z) :- not(h(Z)).
h(a).
```

What should our response be to the query $g(z)$?

The problem is that the query generates the subgoal $\neg h(z)$ and conventional PROLOG interpreters are incapable of dealing with negative subgoals that contain unbound variables. (Such subgoals are called *floundered.*) Two solutions have been proposed.

The first, due to Chan [6], is to respond to the query $g(z)$ with the answer, "all z except for $z = a$." Chan proposes implementing this suggestion by maintaining and manipulating a list of possible exceptions to a particular query; we will not repeat the details here.

We can capture this in our setting by using as our set of truth values not simply the four-point bilattice $F = \{t, f, u, \bot\}$ but the set of *functions* from possible bindings into F. Thus the truth value to be assigned to our query $g(z)$ should be the function that maps all z except $z = a$ into true, and maps $z = a$ into unknown.

The set of functions from binding lists into F inherits a bilattice structure from the existing structure on F itself; the bilattice operations are all computed pointwise. Thus if f_1 and f_2 are two such functions and σ is a binding list,

$$(f_1 \wedge f_2)(\sigma) = f_1(\sigma) \wedge f_2(\sigma)$$

Given this, how does the analysis of our example proceed? The declarative translation of the logic program is

$$\neg\Box h(z) \supset g(z)$$
$$h(a)$$

and the first phase looks like this:

	U	E	J
1	$g(z)$		
2	$h(z)$	$g(z)$	$\langle g(z), t, \neg\Box h(z)\rangle$
3	$\emptyset$	$h(z)$	$\langle h(z), t \text{ if } z = a, \emptyset\rangle$

The second phase is:

	R	q	$\phi^J(q)$
1	$g(z), h(z)$		
2	$g(z)$	$h(z)$	t if $z = a$
3	$\emptyset$	$g(z)$	t unless $z = a$

In the final line above, we have that since the truth value assigned to $h(z)$ is true at $z = a$ and unknown elsewhere, the truth value assigned to $\Box h(z)$ is true at $z = a$ and false elsewhere. The truth value assigned to $\neg\Box h(z)$ is therefore false at $z = a$ and true elsewhere.

Proposition 3.32 *Using the functional truth values described above, the result returned in response to a query q is the same as that computed by Chan's algorithm.*

Chan needs to require that there are no cycles in the logic program being considered; our definition of stratification covers this restriction naturally.

We have proposed a different approach to this problem [38], arguing that the exceptions to $g(z)$ may be either difficult to compute or infinite in number. As a result, we may want to respond to the query $g(z)$ simply by reporting, "yes, except for specific choices for z." A formalization of this notion proposes, roughly speaking, that we should be prepared to return an answer without considering possible exceptions if we know that every exception involves binding some variable in the answer [38].

In the example we have been considering, this means that the empty binding is a legitimate response to our original query, since the only exception binds z to a. To see the value of this, suppose that we change the example to the following:

```
g(Z) :- not(h(Z)).
h(a) :- not(j).
```

where j is a sentence the truth value of which is difficult to determine. Investigation of the query $g(z)$ now forces us to decide whether or not j is true, although we may not be interested in this exception to our conclusion that g holds in general.

We formalize this by introducing a new modal operator X on our bilattice of functional truth values; X stands for "ignore exceptions." For a functional truth value f, here is the definition of X:

$$X(f)(\sigma) = \vee_{\chi \leq \sigma} f(\chi)$$

This probably needs some explanation. Since f, the bilattice argument of X, is a function on binding lists, $X(f)$ will be a function on binding lists as well. The value taken by $X(f)$ on a particular binding list σ is the disjunction of the values taken by f on all binding lists that are less specific than σ; we have written $\chi \leq \sigma$ to indicate that a specific binding list χ satisfies this criterion.[19]

In the example we are considering, the truth value eventually assigned to g is the following function, where we assume that the attempt to prove j fails:

σ	$g(\sigma)$
$\emptyset$	t
$z = a$	u

(43)

In other words, the empty binding is mapped to true, since we can prove $g(z)$ for general z. But if z is bound to a, the proof fails because $\Box j$ is false and $h(a)$ is therefore true.

It follows from (43) that the value assigned to the modal expression $X(g)$ is given by:

σ	$g(\sigma)$	$X(g)(\sigma)$
$\emptyset$	t	t
$z = a$	u	t

The reason for the change in the last row is that $z = a$ is more specific than the empty binding, and we disjoin the values taken at both bindings when computing $X(g)$. In other words, $\text{cl}(\phi)(X(g))$ is simply true.

[19] We are saying that one binding list is less specific than another if it binds fewer variables, or binds variables to less specific values.

Proposition 3.33 *Suppose that we have a database ϕ and a query q. Then the value computed by Procedure 3.25 in response to the query $X(q)$ will be the same as the value computed by Ginsberg's algorithm [38] for dealing with floundered subgoals.*

In addition, there is a natural connection between the details of the earlier algorithm and the notion of relevance presented in Definition 3.24. In the example we are considering, investigation of $X(g)$ leads to the following:

	U	E	J
1	$g(z)$		
2	$h(z)$	$g(z)$	$\langle g(z), t, \neg\Box h(z)\rangle$
3	j	$h(z)$	$\langle h(z), t \text{ if } z = a, \neg\Box j\rangle$

(44)

But now j is not relevant to the original query $X(g)$ (although it would be relevant to the query g), and will therefore not be considered further. In fact, a careful examination of the previous algorithm [38] shows that the details of this algorithm are designed to achieve precisely the computational benefits that arise from the use of Definition 3.24.

Step logics We end this section not with another example, but with a comparison between our work and the most successful existing approach to anytime declarative reasoning, Elgot-Drapkin and Perlis' *step logic* [18, 19]. In step logic, one arranges to be able to interrupt a reasoning system after it has taken a certain number of inference steps en route to reaching a conclusion about a particular query.

The fundamental difference between this approach and ours is one of syntax as opposed to semantics. The interruption points used by step logics are necessarily syntactic objects; what constitutes a single inference using one rendering of a declarative database may correspond to many inferences if a different description is employed.

In the approach that we have described, the interruption markers are not inference steps but modal operators – objects that are fundamentally semantic as opposed to syntactic in nature. The definitions we have given (such as the original Definition 3.15 of stratification) appear to be appealing to syntactic features of our database, but this is an illusion; since the fundamental building blocks of our system are semantic, the overall description is also.

As an example, it is not difficult to show that Definition 3.28 will label a truth assignment as semantically stratified whenever it has a unique k-minimal closed and grounded extension. This is clearly a semantic notion and corresponds to the nonmonotonic reasoning community's idea of a declarative theory having a unique extension.

Granted, the truth-functional semantics we have given to modal operators is not the standard one. Nevertheless, modal operators remain semantic objects within our system, and the points at which we interrupt our reasoning are semantic as a result.

Is this a strength or a weakness? It might seem that the location of interruption points is really a metalevel issue; using either a syntactic description or an explicitly metalevel semantic description would allow us to discriminate between interruption markers and the base-level information already present in our system. But perhaps not: The argument can certainly be made that the points at which one wants to interrupt the conventional semantics of a declarative database should be the same as the points at which one also wants to interrupt the associated inference engine.

3.1.5 Summary

Our aim in this section has been to suggest that modal operators can improve the usefulness of declarative systems by marking points at which inference can be suspended and an approximate answer returned. Viewing the modal operators truth-functionally, their correct evaluation requires a recursive call to the theorem prover on the sentential arguments of any particular modal expression; if the computational resources needed by this recursive call are not available, we can instead use the values found by simply searching the database for these sentences.

Using this ideà, we developed an anytime declarative procedure that responds to queries by repeatedly calling a conventional first-order theorem prover and then accumulating the results. Approximate answers are computed quickly and gradually refined if time permits. Although the final answer returned by the system may appear to be changing in a non-systematic way, the quality of the answer improves uniformly as additional information is considered.

3.2 Declarative descriptions of planning domains

Existing planning systems can be grouped into three broad categories: expert planners, general-purpose planners, and declarative planners.

Expert planners, of which there are many, are essentially applications of expert-systems technology to planning problems. The situation in which a particular agent finds itself is classified to determine which of a predetermined set of actions is most likely to be effective in achieving the agent's goals. There has been some interest in constructing the expert decision rules automatically [16], but the approach itself must inevitably be limited by the fact that the agent involved has no real idea what's going on – it is simply mindlessly applying rules that govern its behavior. The ensuing brittleness is typical of expert systems generally.

General-purpose planners, of which there are few, attempt to address this difficulty by working with a set of action descriptors that describe the possible actions in some domain, and then constructing a plan to achieve a particular goal using methods that are independent of the domain in which the agent finds itself. This work began with STRIPS [25]; the most successful existing planners of this sort are arguably SIPE [106] and O-PLAN [9].

There are two difficulties with the general-purpose approach. The first is that the computational complexity of planning problems is typically very high, making it impractical to generate a complete plan that is guaranteed to achieve a particular goal. SIPE and O-PLAN address this difficulty by restricting the form of the actions they can consider and by allowing the user to provide such specific control information that he is, at least in some sense, actually *providing* the system with the plan that he wishes to see constructed.

The nondeclarative restrictions placed on the form of the actions being considered unfortunately make these planners nonuniversal; there are domains for which any particular restriction is inappropriate. This is the essence of the second difficulty: General-purpose planners, by committing at a fundamental level to a specific description of actions, inherit some (but by no means all) of the brittleness of their expert-planning predecessors. "General-purpose" planners are only general-purpose within the bounds established by assumptions embodied in the form of the action descriptors.

Declarative planners (of which there are none) attempt to address these difficulties by viewing planning as a purely declarative activity. More specifically, declarative planners view planning as theorem proving set against the background of a declarative system that describes actions in a particular domain.

This idea is an old one, dating back to Green's QA3 system [50]; as work on declarative systems generally has advanced, the attractiveness of the approach has remained. With the development of nonmonotonic reasoning, for example, it was suggested that this general declarative notion could be applied to construct a planner that would be able to jump to conclusions while building its plans. It was later suggested that assumption-based truth maintenance [10], another general declarative technique, might bear on the problem of debugging plans that appear to be nonmonotonically sound but that closer inspection reveals to be flawed in some way [36].

One reason that there are no established planners of this type is that the underlying declarative descriptions of action are themselves lacking. The best-known example of this is the infamous Yale shooting problem [53], although a variety of researchers have found solutions to this particular difficulty.

A more fundamental problem with declarative descriptions of action is that they are simply unsuitable for inclusion in planners. The approach introduced with QA3 [50] and reiterated subsequently [31] is still a valid one – given a monotonic description of a domain, it is indeed possible to view planning as theorem proving. The difficulty is that it is not practical to do so.

The reason for this can be seen by considering the frame axiom. Here is a typical rendering of it that might be included in a nonmonotonic axiomatization:

$$\texttt{holds}(f,s) \wedge \neg\texttt{ab}(a,f,s) \supset \texttt{holds}(f,\texttt{result}(a,s)) \tag{45}$$

Informally, this axiom says that if some fluent f holds in a situation s and the action a is not abnormal in that it reverses f when executed in the situation s, then f will continue to hold after the action is completed.[20]

There are technical problems with this definition, but they can be avoided [4, and many others]. But an overwhelming *computational* difficulty can be seen if we imagine using (45) to propagate a set of fluents through a long sequence of actions. The application of (45) for each action and to each fluent will result in a prohibitively large number of consistency

[20] By a *fluent*, we will mean something that can be true or false in a particular situation, such as `wearing(fred,parachute)` saying that Fred is wearing his parachute.

checks, making the system unusable in practice.[21]

This problem is avoided in general-purpose planning systems by using a nondeclarative description of action that has more attractive computational properties. In STRIPS, for example, actions are described in terms of add and delete lists, reducing the complexity of the reasoning enormously. The STRIPS formalism cannot deal with the inferred consequences of actions, however [63]. Partial solutions to this difficulty continue to describe actions in nondeclarative terms [49].

The intellectual foundation for the work described in this section of the report lies in an attempt to present a declarative description of this existing STRIPS-based work; we have tried to develop a formalization of action that will be computationally viable in the situations likely to arise in planning. The two specific heuristic commitments that we will make are the following:

First, we will assume that fluents typically survive long sequences of actions before being needed; a robot should be able to put a wrench in its toolbox, perform most of its day's activities, and conclude at a single stroke that the wrench is still in the toolbox. We will describe this by saying that our formalization of action needs to be *event-driven* in the sense that propagating fluent values through idle periods not incur significant computational expense.

Second, we will commit ourselves to a system that can reason about actions in an anytime fashion. When asked the value of a fluent in a specific situation, the system must produce some answer quickly, modifying that answer as necessary if allowed to consider more subtle features of the situation involved. It is generally recognized that planning problems are sufficiently difficult that approximate answers are inevitable; we are simply requiring that this sort of computational response be present in the description of action that underlies the planner itself. These observations are natural successors to our general discussion of the real-time properties of declarative systems in the last section.

The reason that we have chosen to discuss these two problems is not that there are no others (there are), but that the solutions to them are linked. Roughly speaking, both difficulties can be addressed by taking the truth value assigned to a sentence to be not single values such as "true" or "false," but functions from a set of time points into such values.

This approach leads to an event-driven description because it allows us to conveniently describe the expected future behavior of fluents in a compact fashion. Instead of saying, "The wrench is in the toolbox at 9:15," and, "Things in toolboxes tend to remain there," we can simply say, "The wrench is expected to be in the toolbox for the rest of the day," meaning that the truth value assigned to the sentence

$$\mathtt{in(wrench, toolbox)}$$

is a function that maps the entire temporal interval from 9:15 to 5:00 to the value t (or perhaps dt if we are prepared to admit the possibility of subsequent information reversing our conclusions). The problem of making our description event-driven now becomes essentially

[21] Unlike Section 2, we are currently viewing the default (45) as base-level knowledge and not as implicit control information.

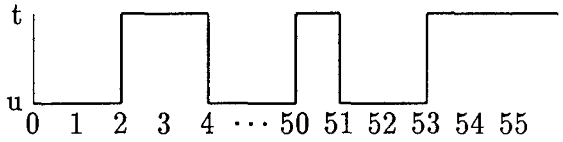


Figure 6: A fluent that is true three times

a matter of finding a data structure for the functional truth values that efficiently encodes the behavior of fluents that change only infrequently.

The idea of taking truth values to be temporal functions also bears on our requirement that the implementation of our formalism exhibit anytime behavior. As an example, consider a sentence such as, "One second after the valve is closed, the pressure will increase," which we will write somewhat schematically as

$$\texttt{delay(closed-valve)} \supset \texttt{pressure} \tag{46}$$

where `delay` is an operator that we will use to push the temporal description of the valve into the future.

From a formal point of view, the `delay` operator appearing in (46) is modal, since one of its arguments is not an object in our language but a declarative sentence instead. In (46), the modal operator `delay` corresponds to the function `delay` that is given by

$$[\texttt{delay}(f)](t+1) = f(t) \tag{47}$$

Note that `delay` accepts a function as its first argument and returns a function as its result, since the truth values that we are using are themselves functional. We saw in Section 3.1 that this interpretation leads to anytime behavior.

3.2.1 Products of bilattices

If B is a bilattice, then B^2, the collection of ordered pairs of elements of B, inherits a bilattice structure from B where all of the bilattice operations are computed pointwise. (The construction is analogous to the construction of the Cartesian plane $\mathbb{R}^2$ as the product of two copies of the real line.) More generally, for any set S, the set B^S of functions from S into B inherits a bilattice structure from the set B.[22]

It follows that if we have some set T of time points, then the set B^T of functions from T into the "base" set of truth values B has the structure required of a set of truth values. As an example, if we take T to be the integers, then the graph in Figure 6 shows the truth value assigned to a fluent that is true for two units of time at $t = 2$, for one unit of time at $t = 50$, and for all time after $t = 53$.

Our event-based philosophy now corresponds simply to a data structure that represents these truth values by listing the points at which the value changes. In Figure 6, for example,

[22] There is no real difference between viewing the set B^2 as the set of ordered pairs of elements of B, or as the set of functions from the two-point set $\{1, 2\}$ into B.

we record the fact that the fluent is unknown at time 0, true at time 2, unknown at time 4, and so on; values at a total of six points are recorded. Determining the value of the fluent at any intermediate time t is a matter of walking along the graph until the next event is later than t, and taking the value at the last time encountered. Note that the computational effort required to determine the value of the fluent in Figure 6 is completely independent of the length of the gap between times 4 and 50.

Extensions The ontological shift that we are proposing does not in and of itself commit us to any specific computational or representational strategy. As an example, the simple representation scheme that we described in the previous paragraph can easily be extended in a variety of ways:

1. The set T of time points only needs the structure of a partial order in order for the above approach to work; to determine the value of a fluent f at some particular time t, we walk our way along the function until we find ourselves between two points t_0 and t_1 such that
$$t_0 \leq t < t_1,$$
so that t is no earlier than t_0 and t_1 is later than t. The value of f at t is then the value taken at t_0.

2. As an example, taking the above partial order to be the continuous real line allows us to avoid our earlier implicit assumption that time was discrete. This particular choice commits us to fluents being true over half-open intervals $[x, y)$ only, but this can be avoided by introducing auxiliary elements x^+ for each $x \in \mathbb{R}$ such that the half-open interval $[x, y^+)$ in fact denotes the closed interval $[x, y]$.

3. Another example involves taking the elements of the partial order to be action sequences, where an action sequence a_2 temporally follows a sequence a_1 whenever a_2 is an extension of a_1. Nonlinear action sequences can be handled by weakening the partial order to cater to possible linear action sequences consistent with a given nonlinear one. Alternatively, by using sequence variables to denote arbitrary sequences of actions following some particular one, it is possible to use the partial order given by variable instantiation to capture this temporal information. This general idea is central to the development of the approximate planning system discussed in the next section.

4. It is also possible to extend the scheme by introducing "decay functions" that describe how a fluent's truth value is expected to change as time goes by. In Figure 6, for example, the fluent's truth values do not change at all as time passes; a more realistic example might involve the truth value of the fluent falling from t to dt at times 3 and 54, as shown in Figure 7. Here, our confidence in the truth of the fluent decays as time passes, corresponding to the application of a nonmonotonic frame axiom. As before, information is recorded only when the truth value of the fluent changes from the expected one, so we still need to record information only about the "events" at

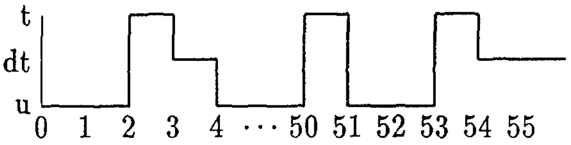
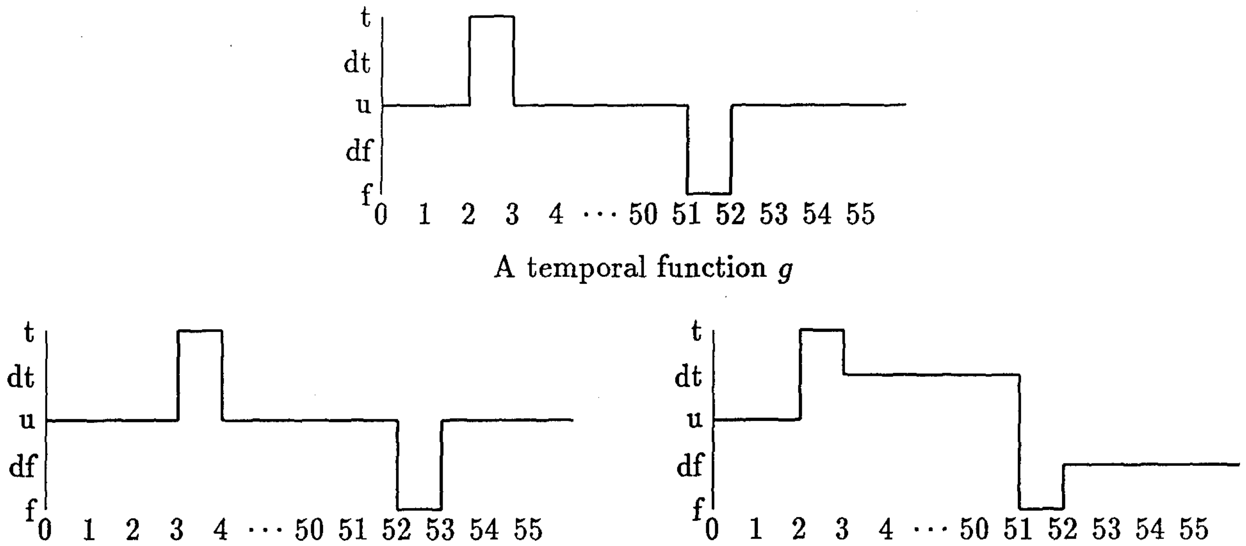


Figure 7: A default frame axiom



A temporal function g

`delay`(g)

`propagate`(g)

Figure 8: Delay and `propagate`

times 0, 2, 4, 50, 51 and 53. By changing the set of base truth values to which the temporal functions map, this idea can be extended to include a wide variety of temporal behaviors, such as probabilistic decay functions [12]. A fuller discussion of this point can be found elsewhere [39].

The computational efficacy of the scheme that we have proposed is preserved in all of these cases.

3.2.2 Temporal modalities

There are two modal operators needed to describe action in this setting. The first separates the occurrence of an action from its effects; if an action occurs at time t_0, the effects will presumably not appear until time $t_0 + 1$. This is the `delay` operator of Section 3.2.1.

As with the modal operator X of the previous section, `delay` accepts a function as its argument and returns a function as its result. Figure 8 shows the result of applying `delay` to a temporal function that changes from true to false.

A second modal operator is needed to describe the way in which fluents persist from one moment to the next after they are inserted into our declarative database. In nondeclarative systems, this problem is solved using what Chapman calls a *modal truth criterion* [8]. A modal truth criterion combines information about the validity of fluents at one time with with information about action occurrence to determine whether or not a particular fluent holds at some subsequent time. The modal truth criterion is aptly named: It does indeed correspond not to an axiom, but to a modality.

More specifically, suppose that $g(t)$ indicates those time points at which a particular fluent λ is either established (i.e., made true) or disestablished (made false). We will now assume that there is some modal operator `propagate` such that $\texttt{propagate}(g(t))$ is the truth value of the fluent itself, presumably persisting from its points of establishment into the future by virtue of the frame axiom. Figure 8 also shows the result of applying `propagate` to the temporal function g. Between times 3 and 51, the result is true by default; after time 52, it is false by default.

From the two modal operators `delay` and `propagate`, we can build an anytime description of action. If a is an action that causes a fluent λ to be true in a persistent way (such as putting a tool in a toolbox), we write[23]

$$\texttt{propagate}(\texttt{delay}(a)) \supset \lambda \tag{48}$$

while if a causes λ to be true only instantaneously (like striking a match causing a light), we write simply

$$\texttt{delay}(a) \supset \lambda \tag{49}$$

Note that the future behavior of any particular fluent is determined not by applying some blanket frame axiom, but instead by an axiom like (48) or (49) describing this future behavior when the fluent is first asserted (i.e., when the robot puts a tool in the toolbox or strikes a match). This is an important distinction between our approach and the conventional one; we prefer our ideas because it is *not* the case that the frame axiom applies to all, or even most of the fluents we encounter in everyday life.

As an example, consider the problem of entering a crosswalk when there is a car five feet away approaching at 60 MPH. Are we to apply the frame axiom to the fact that the car is five feet from the crosswalk, or to the fact that it is moving at a high speed? Clearly to the latter, although there is no information in a blanket frame axiom indicating that we should do so.

This is related to the well-known *problem of induction* [92]. How is it that we know to apply the frame axiom to a predicate such as "blue" or "green" but not to one such as "grue"

[23] The axiom (48) is not quite satisfactory as it stands, because it is awkward to combine it with domain constraints describing ramifications of the action in question. This is handled by reifying the fluents so that they can be treated as objects in our language, and replacing (48) with an axiom like

$$\texttt{causes-persistently}(a, \lambda) \wedge \texttt{propagate}(\texttt{delay}(a)) \supset \texttt{holds}(\lambda)$$

A complete axiomatization appears later in this section.

(green until July 10th but blue subsequently) or "bleen?" Although we have not provided an answer to this question, we have indicated clearly the declarative point at which such an answer is used – in the description of the expected future behavior of newly established fluents. Similar observations have also been made by Myers and Smith [71].[24]

There are two other points to be made about this example. The first involves the modal operator `delay`; the second, the impact of relevance notions in this setting.

The point we would like to make about `delay` is that it is monotonic. From a formal point of view, this means that it cannot serve as an interruption marker unless it is equipped with artificial groundedness conditions. The impact of such groundedness conditions is that although we will remain able to reason from a sentence p (saying, for example, that an action occurred at some specific time) to the sentence $\mathtt{delay}(p)$ (giving the effects of the action), we will *not* be able to reason in the other direction. If we observe the effects of an action, we will not be able to conclude that the action occurred because this inference step is not grounded – one is not permitted to draw conclusions about sentential arguments from the truth of a grounded modal expression. There is no obvious solution to the conflict between a natural desire to view `delay` as an interruption marker and a desire to reason backwards through this particular modality.

With regard to relevance, the implications can be quite subtle. Imagine that our goal is to show that for some sentence p, $\mathtt{propagate}(p)$ holds at time 54. There is clearly no point to considering arguments that lead to the conclusion that p is true at a time later than 54, since `propagate` will have no useful effect when applied to such a truth value.

Ignoring times later than 54, suppose that we manage to show that p was inserted into the database at time 2 (as in the figure), so that $\mathtt{propagate}(p)$ now holds at time 54 by virtue of the frame assumption. Now there is no reason to find other points at which p is inserted into the database; the information we already have suffices. But there *is* reason to search for times at which p is removed from the database, provided that these times are no later than time 54. This is the consistency check in our setting.

Continuing as in the diagram, suppose that we find that p is disestablished at time 52. Now there is no point in showing that it is reestablished at any time other than 53 or 54, and so on.

3.2.3 An example

We now present a complete axiomatization of a STRIPS-like theory. Once again, the strength of our methods lies not only in our ability to formalize existing planners, but also in our ability to use declarative descriptions of action. We can easily modify our description to include actions of variable duration, conditional effects and so on.

We take the bilattice of base truth values to be the four-point bilattice F, and use three separate causal predicates:

1. $\mathtt{causes}(a,p)$ means that the action a causes the fluent p to be true instantaneously.

[24] Other authors have also argued against blanket frame axioms, suggesting instead that one should reason from specific fluent changes to the occurrence of an action of some sort [51, 79, 85].

2. adds(a, p) means that the action causes the fluent to be true in a way that is expected to persist into the future.

3. deletes(a, p) means that the actions causes the fluent to be false in a way that is expected to persist into the future.

Fluents are reified using a holds predicate; the insertion of the reified fluents into the database is also reified using a triggers predicate, so that holds is the result of applying the modal operator propagate to triggers.

Here are the axioms associated with these predicates, expressed in a PROLOG-like style:

```
holds(P) :- causes(A,P), delay(succeeds(A)).
holds(P) :- propagate(triggers(P)).
triggers(P) :- adds(A,P), delay(succeeds(A)).
not(triggers(P)) :- deletes(A,P), delay(succeeds(A)).
```

Our notation here involves a first-order negation operator as opposed to the negation-as-failure operator used in PROLOG. The not appearing in the head of the last of the above rules is first-order negation.

An action succeeds if it occurs and all its preconditions are satisfied:

```
succeeds(A) :- occurs(A), prec(A,P), all-hold(P).
all-hold([]).
all-hold([X|Y]) :- holds(X), all-hold(Y).
```

Finally, there is a dummy action init that takes place at time 0 and is used to construct the initial situation. This action has no preconditions.

```
occurs(init).             ; true at time 0 only
prec(init,[]).
```

To see our formal ideas at work, let us look at a simple domain where Fred, instead of falling out of an airplane, is the target of an assassination attempt. (The example bears a passing resemblance to the infamous Yale shooting problem [53].) This domain has three actions, load, unload and shoot, all of which involve the gun being used in the murder. Shooting Fred has as a precondition that the gun be loaded:

```
prec(load,[]).
adds(load,loaded).

prec(unload,[]).
deletes(load,loaded).

prec(shoot,[loaded]).
deletes(shoot,alive).
```

In the initial situation, Fred is alive and the gun is loaded; both facts are expected to persist into the future:

```
    adds(init,alive).
    adds(init,loaded).
```

Fred's security guard comes by at time 1 and unloads the gun, but it is reloaded at time 10. The assassin shows up at time 20:

```
    occurs(unload).         ; true at time 1 only
    occurs(load).           ; true at time 10
    occurs(shoot).          ; true at time 20
```

When we indicate "true at time t," we mean that these facts are inserted into the database with truth values indicating that they are true at these times. Recall that our responsibility in providing the input theory is to provide a truth assignment in the sense of Definition 3.2, as opposed to simply a list of true sentences (i.e., sentences true at all times).

Here is how the analysis proceeds. The mnemonics are: a is the query, **holds(alive)**; p is **propagate** and d is **delay**. We use subscripted s for actions' succeeding; s_i is **init** succeeding, s_s is **shoot** succeeding and s_l and s_u are `load` and **unload** succeeding. In a similar way, t_a is **triggers(alive)** and t_l is **triggers(loaded)**. At each point in the following analysis, we also indicate the current value that would be taken by **holds(alive)** were we to compute ϕ^J:

	U	E	J	$\phi^J(a)$
1	a			u
2	t_a	a	$\langle a, t, p(t_a)\rangle$	u
3	s_i, t_a		$\langle t_a, t, d(s_i)\rangle$	u
4	t_a	s_i	$\langle s_i, t \text{ at } 0, \emptyset\rangle$	t
5	s_s	t_a	$\langle \neg t_a, t, d(s_s)\rangle$	t
6	t_l	s_s	$\langle s_s, t \text{ at } 20, p(t_l)\rangle$	t
7	t_l		$\langle t_l, t, d(s_i)\rangle$	until time 21
8	s_u, t_l		$\langle \neg t_l, t, d(s_u)\rangle$	until time 21
9	t_l	s_u	$\langle s_u, t \text{ at } 1, \emptyset\rangle$	t
10	s_l	t_l	$\langle t_l, t, d(s_l)\rangle$	t
11	$\emptyset$	s_l	$\langle s_l, t \text{ at } 10, \emptyset\rangle$	until time 21

(50)

The details are as follows:

1. Initially, the goal is simply a: What can we say about holds(alive)?

2. We can prove a by finding times at which Fred's being alive is triggered (i.e., times at which t_a is true), and propagating the result. So a is a consequence of $p(t_a)$. Since t_a itself is currently unknown, a remains unknown. There are no other ways to derive a, so it is moved to E.

3. We can derive `triggers(alive)` by using the fact that `init` triggers Fred's being alive; this is how we establish information about the initial state. We have not yet considered proofs of $\neg t_a$; a itself remains unknown because we have not yet assigned a truth value to `succeeds(init)`.

4. The single proof of s_i (the success of the dummy action `init`) tells us that `init` succeeds at time 0; no modalities are used. We move s_i to E and can also conclude that

   `delay(succeeds(init))`

   is true at time 1, so that

   `propagate[delay(succeeds(init))]`

   is true at all times. Fred is always alive.

5. Now we look for proofs of $\neg t_a$. What can cause Fred to become not alive? The only proof found involves the assassination succeeding, so we add s_s to U.

6. The shooting will succeed at time 20, provided that the gun is loaded at that time. This is the information in the justification triple added in this step of the analysis; we add t_l to U and continue.

7. One way to trigger the gun's being loaded is to realize that `init` does so, since the gun is loaded in the initial situation. But we have already considered s_i, which is in E at this point. We conclude from this that the gun will be loaded at time 20, that the assassination attempt will succeed, and therefore that Fred is alive only until time 21.

8. We continue to look for new proofs of t_l or $\neg t_l$; in fact, we look for proofs of $\neg t_l$ only since a new proof of t_l would not be relevant to our overall conclusion. We see that $\neg t_l$ is a consequence of the `unload` action succeeding, so we add s_u to U.

9. We next note that s_u is true at time 1, so that the gun is presumably unloaded from time 2 on, the shooting now fails, and Fred is believed to survive after all.

10. At this point, t_l is relevant once again, and a new proof is found from the information in the database – instead of relying on the initial situation to load the gun, we look for an explicit `load` action.

11. There is a successful `load` action at time 10. The gun is loaded when the assassination is attempted and Fred dies at time 21. At this point U is empty and the analysis is complete.

Note the natural way in which our conclusions change. Initially, we don't know whether Fred is alive or not, and we begin by applying the frame assumption to conclude that he is and is likely to remain so. We then conclude that the assassin will succeed in killing him by a similar application of the frame axiom to the gun's being loaded in the initial situation.

Further consideration reverses this conclusion when we realize that the security guard's unloading the gun invalidates this reasoning. But the gun is reloaded at time 10, and we eventually conclude that the shooting succeeds after all. An alternative line of reasoning, more stable in this particular instance, would have us conclude initially that the gun was loaded not because of the initial `init` action, but because of the explicit loading action at time 10. This leads to an argument that is not subsequently defeated by other information in the database.

In this example, we can clearly see the two features that have been our focus in this section - the event-driven nature of the description, evidenced by the lack of computational effort devoted to the "idle" time between $t = 10$ and $t = 20$, and the anytime nature of the analysis, shown in the shifting conclusions displayed in (50).

3.2.4 Summary

We have argued that a declarative description of action should have two properties. First, it should use truth values that directly capture the complete history of fluents that change over time; second, it should manipulate of these truth values modally. An implementation of our ideas correctly analyzes a simple example similar to the shooting scenario presented in [53], but the theoretical justifications for this approach are more compelling:

1. The approach embeds the action descriptions in a full declarative language, and investigates the consequences of actions by proving theorems against this background. A system developed in this way will benefit from developments elsewhere in the theorem-proving community in a way that a more *ad hoc* approach cannot.

2. The event-driven nature of the approach allows us to reason in a computationally viable way about fluents that change value only infrequently. As discussed earlier, we do this without committing ourselves to any specific ontology of time or of action.

3. The natural implementation of our ideas exhibits an anytime behavior that we can expect to be present in the associated planning system as well. Furthermore, this ability to incrementally refine our conclusions is grounded in a solid formal foundation.

4. Finally, the approach we have described avoids the use of a blanket frame axiom such as (45), which is likely to fall prey to the problem of induction. Although we have presented no solution to this difficulty, our approach makes clear that such a solution will need to be reflected in the declarative description of our domain, since the expected future history of any particular fluent needs to be asserted when the fluent itself is added to the database.

3.3 Approximate planning and planning for subgoals separately

We are finally ready to turn our attention away from descriptions of action generally and toward the planning problem specifically. Our overall aim is to develop a planner capable of both using the declarative formalization we have developed and of reasoning about subgoals separately.

We begin, however, with a discussion of the planning problem itself. Constructing a planner with the properties we desire involves redefining this problem somewhat, and we will motivate this redefinition in terms of a modified description of plans.

When we talk about a plan for achieving a goal, we typically mean not one plan but many. As an example, if I say on Thanksgiving that my plan for preparing a turkey involves stuffing and roasting it, I hardly mean that these are the only actions I will take between now and when the turkey is done. I may also plan on making sweet potatoes and pumpkin pie, buying a bottle of wine, calling family members to wish them happy holidays, and other actions even further removed from my stated goal of turkey preparation.

In fact, my plan "stuff the turkey and then roast it" might be represented something like this:

$$[\ldots\texttt{stuff}\ldots\texttt{roast}\ldots] \qquad (51)$$

where the ellipses denote currently undetermined action sequences that I might intersperse into the above plan. If I need to roast the turkey *immediately* after stuffing it, I might write that as

$$[\ldots\texttt{stuff roast}\ldots] \qquad (52)$$

where the second set of ellipses has been dropped.

There are, of course, many instance of (51) that are unsatisfactory. Perhaps I run the turkey through a paper shredder before beginning preparation, or unstuff it after stuffing it, or garnish it liberally with peanut butter before serving. In what sense can we say that (51) is our plan when so many things can go wrong?

The conventional approach to this problem is to deal not with plans such as that appearing in (51), but with far more specific plans such as

$$[\texttt{stuff yams telephone roast eat}] \qquad (53)$$

where there are guaranteed to be no extraneous actions that might interfere with our achieving our goal. But from a practical point of view, the plan (53) is nearly worthless, since it is almost inconceivable that I execute it exactly as written.

There are many other examples of this phenomenon. If we intend to construct plans by retrieving them from a library of known solutions to similar problems (so-called *case-based* planning [52]), it is important that the plans in the library include some measure of flexibility. After all, it is unlikely that the new situation in which we find ourselves will be an exact match for the situation in which the plan was constructed.

If we are to plan for conjuncts separately and then merge the results, it is essential that the solutions to the individual conjuncts be plan schemas such as (51). At the very least, we need the ellipses as place holders for the actions being merged in from solutions to other conjuncts!

An example of a conjunctive planning problem is shown in Figure 9. The goal is to get A on B and B on C, but there is a restriction to the effect that one cannot build a four-block tower.

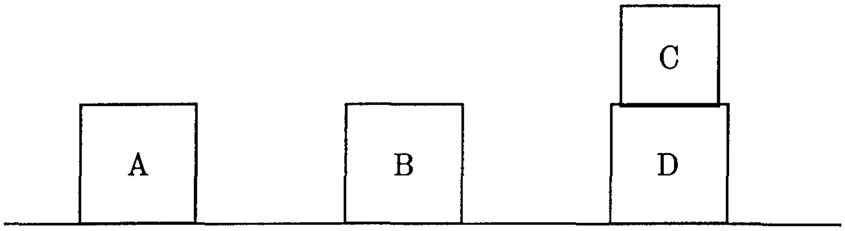


Figure 9: Get A on B on C without building a 4-block tower

For a human planner, the problem is easy. We realize that the general plan for getting B onto C is simply to move it there, and similarly for getting A on B. When we combine these two plans, however, we encounter a problem – the action of moving A to B will fail. We therefore modify the plan for getting B onto C, adding the additional action of moving C to the table.

We presented this problem to the authors of two generative planning systems – Minton (PRODIGY [68]) and Wilkins (SIPE [106]). Both reported (personal communication) that the problem would pose no significant difficulties for them and that they could solve it by adding an additional precondition to the action $\texttt{move}(x, y)$ to the effect that y had to be either on the table or on a block z that was on the table.[25]

The problem with this approach is that it doubles the branching factor for all planning problems. This will lead to prohibitive computational difficulties as the problems involved get larger; imagine having to move a block prior to constructing a 13-block tower in a domain that prohibits 14-block ones. As an example of the immediacy of these difficulties, Penberthy and Weld's UCPOP system [74] proved incapable of solving the 4-block version of the problem in Figure 9 without the inclusion of domain-specific control information.[26]

Worse still is the fact that the branching factor is being increased on *all* problems, not just those that involve tall towers. Imagine, for example, that we can only put a blue block on a red one if the red block is on the table. The branching factor will still be doubled even if we are working in a domain without blue blocks![27]

Explicit control rules provide potential ways around these particular difficulties, but their use is problematic. What control rule are we to use if the previous domain includes painting actions, so that the colors of blocks can change? What control rule would allow us to efficiently solve the problem in Figure 9 if the constraint were changed so that only *five*-block towers were prohibited?

Related problems appear in plan debugging. If a human planner discovers a bug in one portion of a plan to achieve a complex goal, the typical response is to restrict the impact of the bug to a small portion of the analysis and to then plan around the problem. That we can

[25] Wilkins made the alternative suggestion of creating two move operators. This is equivalent in practice, however; doubling the branching factor by introducing a second move operator is equivalent to doubling it by introducing a disjunction into the precondition.

[26] Will Harvey, personal communication. The problem is not one of time, but of space; UCPOP reported that it had exhausted its available memory while working on this problem.

[27] This is assuming that we treat color as a precondition and not as a filter. We would need to do this if there were actions available that changed blocks' colors.

make modifications that address the bug without destroying the effect of the original plan depends on our commonsense ability to construct and manipulate plans like (51) – plans that, while not holding universally, do hold in general.

In this section of the report, we develop a formalization of the ideas that are implicit in the plan (51), and then describe the use of these constructs in conjunctive planning. Please bear with us while we work through the mathematics, since there are a variety of fundamentally new ideas that we need to formalize.

1. We first need to describe plans that can have new actions added to them in arbitrary ways but that can still include the immediacy requirements of a plan such as (52). This is our goal in the next section, where we also present a variety of mathematical results about these new plans that will be needed later.

2. We next need to define conditions under which a plan approximately achieves a goal. The basic idea here is that a plan P is approximately correct if most instances of P that could actually be executed do indeed achieve the goal. We formalize this in Section 3.3.2 by introducing the idea of an exception to a plan and formalizing conditions under which plans hold sufficiently frequently that we are prepared to treat them as approximately correct.

3. The problem of building a planner around these ideas is discussed in Sections 3.3.3 and 3.3.4. Section 3.3.3 discusses the theoretical issues involved in the construction of the planner, showing that it is indeed possible to plan for conjuncts separately using our ideas. Section 3.3.4 discusses the implementation of our work.

Let us also add something of a disclaimer at this point. We do not mean to imply that existing implemented systems are incapable of manipulating expressions such as (51). O-PLAN, for example, appears to use ideas such as these routinely [9, 102]. But planners that behave in this fashion have thus far lacked formal foundation, and correcting *that* is our intention here. In providing a solid formal foundation for nonlinear planning, McAllester and Rosenblitt's paper [65] was both a step forward and a step back; although it formalized many ideas that had previously eluded precise description, it omitted many of the procedural tricks that make implemented planners effective. As a result, formally well-grounded planners such as that described by Penberthy and Weld [74] typically exhibit performance far worse than that of the informal systems that preceded them. Our intention here is to shed some formal light on the ideas that have proven so effective in practice.

3.3.1 Plans

We will adopt the view that a plan is a partially ordered collection of actions, where an action is a functional expression such as $\texttt{move}(a, b)$:

Definition 3.34 *An* action *is either a variable or a functional expression, where the arguments to the function may themselves include variables. A* ground action *is an action that contains no variables.*

By an action such as $\texttt{move}(a, ?)$ we will mean the action of moving a to the location ? where ? will presumably be determined by other considerations.

We cannot now simply define a plan to be a partially ordered sequence of actions, since we need to be able to distinguish between (51) and (52). In some cases, we will want the action a to precede b *immediately*, while in others, there may be many actions interspersed between the two. We handle this as follows:

Definition 3.35 *A* plan *is a triple $(A, \leq, *)$ where A is a finite collection of actions and $\leq$ is a partial order on A; by $a \leq b$ for actions $a, b \in A$ we mean that a must precede b. $*$ is another binary relation on A with $* \subseteq <$, so that whenever $a * b$, $a < b$ as well. We will assume that $*$ and $\leq$ also satisfy the following conditions:*

1. *If $c * a$ and $c < b$, then $a \leq b$.*

2. *If $b * c$ and $a < c$, then $a \leq b$.*

We will take $a * b$ to mean that b is a successor of a for which no intermediate actions are permitted.

Note that the definition refers to A as a collection instead of as a set; this is to allow the same action to appear multiple times in the partial order. To understand the additional conditions, suppose that a is an immediate successor of c, so that $c * a$. Now if b is some other successor of c, then a must precede b as well. The second condition is the dual of the first.

In practice, plans are bounded by initial and terminal actions. We model this by requiring that plans contain dummy initial and terminal actions denoted by d_i and d_t respectively:

Definition 3.36 *A plan $(A, \leq, *)$ will be said to be* bounded *if $d_i \in A$ and $d_t \in A$ with $d_i \leq a$ and $d_t \geq a$ for all $a \in A$.*

We will assume throughout that all plans are bounded.

The general turkey-roasting plan (51) corresponds to the partial order

$$d_i < \texttt{stuff} < \texttt{roast} < d_t$$

while the plan that roasts the turkey immediately after stuffing it corresponds to

$$d_i < \texttt{stuff} * \texttt{roast} < d_t$$

The second inequality has been made an instance of $*$.

Before proceeding, let us spend a moment discussing the difference between $*$, our annotation for the links in a partially-ordered plan, and the causal annotations introduced by McAllester and Rosenblitt [65].

McAllester and Rosenblitt's links serve more a bookkeeping function than anything else; the information they contain (which action is intended to achieve which precondition) could be recovered, if need be, from the plan being constructed. Recording the information on the arcs serves the computational purpose of making the plans more efficient to work with.

Our $*$ annotation is different. Annotating an arc with $*$ makes the associated plan a semantically different object, in the sense that we have added a new constraint to the set of linearizations of the given plan. Note also that our language is fundamentally more flexible than McAllester and Rosenblitt's – we allow the addition of arbitrary new actions to plans, while they do not. This is important, since it enables us to work with the flexible plan (51) instead of the far more restrictive (53).

Lemma 3.37 *If $(A, \leq, *)$ is a plan, then for any $a, b, c \in A$:*[28]

1. *If $a * b$ and $a * c$, then $b = c$.*
2. *If $a * c$ and $b * c$, then $a = b$.*

An action can have at most one immediate predecessor or successor.

Suppose now that we have some plan $(A, \leq, *)$. If there is a chain

$$c = c_0 * \cdots * c_n = a$$

and $c \leq b$, then b must either be one of the c_i's or it must follow a; there is no other "room" between c and a. The following lemmas capture this, where we have written $\overline{*}$ for the transitive closure of $*$:

Lemma 3.38 *Let $(A, \leq, *)$ be a plan. Then for any $a, b, c \in A$:*

1. *If $c \leq b$ and $c \,\overline{*}\, a$, then either $c \,\overline{*}\, b$ or $a \leq b$.*
2. *If $b \leq c$ and $a \,\overline{*}\, c$, then either $b \,\overline{*}\, c$ or $b \leq a$.*

Lemma 3.39 *Let $(A, \leq, *)$ be a plan. Then for any $a, b, c \in A$, if $a \,\overline{*}\, b$ and $a \leq c \leq b$, then $a \,\overline{*}\, c$.*

It is also straightforward to define conditions under which two plans are equivalent or one is an instance of the other:

Definition 3.40 *Two plans P_1 and P_2 will be called* equivalent *if they are identical up to variable renaming.*

Definition 3.41 *A plan $(A_1, \leq_1, *_1)$ is an* instance *of another plan $(A_2, \leq_2, *_2)$ if there is a binding list σ and a 1-1 mapping $i : A_2 \to A_1$ with the following properties:*

1. *For each $a \in A_2$, $i(a) = a|_\sigma$. In other words, the mapping i maps a to an action that is the same as that constructed by applying the bindings in σ to a.*

[28] Proofs appear in [43].

2. $i(\leq_2) \subseteq \leq_1$. *Every ordering constraint on the second plan appears in the first as well.*

3. $i(*_2) \subseteq *_1$.

An example will probably help. If the stuff and roast actions accept an argument but the eat action doesn't,

$$[\ldots \texttt{stuff(turkey) roast(turkey)} \ldots \texttt{eat}] \tag{54}$$

is an instance of

$$[\ldots \texttt{stuff(?) roast(?)} \ldots] \tag{55}$$

The plan (54) corresponds to the partial order

$$d_i < \texttt{stuff(turkey)} * \texttt{roast(turkey)} < \texttt{eat} * d_t \tag{56}$$

while (55) corresponds to

$$d_i < \texttt{stuff(?)} * \texttt{roast(?)} < d_t \tag{57}$$

The required injection is now given by

$$i(x) = \begin{cases} x, & \text{if } x = d_i \text{ or } x = d_t; \\ \texttt{stuff(turkey)}, & \text{if } x = \texttt{stuff(?)}; \\ \texttt{roast(turkey)}, & \text{if } x = \texttt{roast(?)}. \end{cases}$$

and the binding list σ binds ? to turkey.

It is clear that for each action x, $i(x) = x|_\sigma$. The image of $\leq$ in (57) under i is the partial order

$$d_i < \texttt{stuff(turkey)} < \texttt{roast(turkey)} < d_t$$

and is clearly included in the partial order of (56); the image of $*$ under i contains the single pair

$$\texttt{stuff(turkey)} * \texttt{roast(turkey)}$$

and is once again contained in the $*$ of (56).

Proposition 3.42 *The instance relation of Definition 3.41 is a partial order.*

We will write $P_1 \subseteq P_2$ to denote the fact that a plan P_1 is an instance of another plan P_2.

We have been careful in our definitions not to restrict the number of new actions that can be inserted between any two actions of the plan itself. When it comes time to actually execute the plan, however, we will need to select a specific action sequence.

Definition 3.43 *A plan $P = (A, \leq, *)$ will be called* linear *if the following conditions hold:*

1. Every action in A is ground.

2. $\overline{*} = \leq$.

A linear plan that is an instance of a plan P will be called a linearization *of P.*

In other words, a linearization of a plan replaces all of the variables with object constants and selects an ordering of the actions involved that can be derived solely from the immediacy conditions of $*$. This latter condition implies that no additional actions can be added to the plan.

As an example, the linear plan (53) corresponds to the partial order

$$d_i * \texttt{stuff} * \texttt{yams} * \texttt{telephone} * \texttt{roast} * \texttt{eat} * d_t$$

There is no way to add another action to this ordering without having it either precede d_i, follow d_t, or violate the conditions of Definition 3.35.

Lemma 3.44 *If $P = (A, \leq, *)$ is a linear plan, then $\leq$ is a total order.*

Proposition 3.45 *$P_1 \subseteq P_2$ if and only if the set of linearizations of P_1 is a subset of the set of linearizations of P_2.*

Given the above result, it makes sense to think of a plan in terms of its linearizations; each linearization is a way in which we might actually go about executing the actions in the plan.

Definition 3.46 *A* plan set *is a set of linear plans.*

3.3.2 Approximate correctness

Given now that there will almost inevitably be mistakes we can make, in the sense that there are linearizations of a given plan that do not actually achieve our intended result, how can we formalize the idea that the plan in (51) is correct "in general"?

The solution we will use is an extension of the modal operator appearing in the discussion of negative subgoals in Section 3.1.4. As an example, suppose that we are trying to get block A onto block B in the blocks world. A is clear, but C is currently on top of B and a wide variety of other blocks are scattered around the table. Here is our plan for achieving the goal:

$$[\texttt{move}(C, ?)\ \texttt{move}(A, B)] \tag{58}$$

We plan to move C out of the way, and then move A onto B. We've assumed just for the moment that no additional actions can be added; our interest here involves the variable ? that appears in (58).

Note that there is one location to which we should *not* move the block currently on top of B – if we relocate it to A, B will become clear but A no longer will be. Given this, in what sense is (58) a solution to our problem?

It is a solution in that there are many places to which we can move C, and the one that doesn't work is in some sense pathological – most locations *do* work. What we need to do is to capture the way in which the set of exceptions is small relative to the set of possibilities.

From a formal point of view, the exception involves a specific binding for the variable ?. This leads us to the following:

Definition 3.47 *Given a binding list σ and a plan $P = (A, \leq, *)$, the result of* applying σ *to P is defined to be that plan where the actions in A have had the binding list applied to them but the plan is otherwise unchanged. This plan will be denoted $P|_\sigma$.*

We can, for example, bind ? to A in (58) to obtain

$$[\texttt{move}(C, A)\ \texttt{move}(A, B)]$$

The following result is obvious:

Lemma 3.48 *Given a plan P and binding list σ, $P|_\sigma \subseteq P$.*

We are now in a position to describe conditions under which one set of plans is "small" relative to another. We need to be careful, however, since plans generally have infinitely many linearizations and we can't simply say that Q is small relative to P if Q has many fewer linearizations than P does. Instead, we will say that Q is small relative to P if $Q = P|_\sigma$ but $Q \neq P$. The motivation behind this definition is that there are generally many ways to bind any particular variable and Q is committed to a specific choice.

In the following definition, we will say that Q is *of measure 0 in P* instead of simply saying that Q is small relative to P. The term is borrowed from real analysis, and we use it because the formal definition of smallness has many of the same properties as does the analytic definition on which it is modelled. (The finite union of small sets is small, for example.) The 0 means that the ratio of the size of Q to that of P is approximately 0; we will also say that Q is "of measure 1" in P if this ratio is approximately 1, so that Q and P are comparably sized.

Definition 3.49 *A plan Q will be said to be* of measure 0 *in a plan P if $Q \neq P$ but $Q = P|_\sigma$ for some binding list σ. A plan or plan set Q will also be said to be* of measure 0 *in a plan or plan set P if either of the following conditions is satisfied:*

1. *Q is the finite union of sets of measure 0 in P.*

2. *There exist plan sets R and S with $Q \subseteq R$ and both R and $S - P$ of measure 0 in S.*

The requirement that $Q \neq P$ ensures that the binding list σ is not trivial.

The second condition in the definition handles cases where Q is a subset of a set of measure 0 in P (take $S = P$), or where, for example, one specific linearization has been removed from P. Adding that linearization back to P should not impact the question of whether Q is of measure 0 in P (take S to be the union of P and the missing linearization).

As an example, the plan

$$[\ldots \texttt{move}(a, b) \ldots] \tag{59}$$

is of measure 0 in the plan

$$[\ldots \texttt{move}(a, ?) \ldots] \tag{60}$$

since the variable ? has been bound to a constant in (59). But the plan

$$[\ldots\texttt{move}(a,b)] \tag{61}$$

is *not* of measure 0 in the plan (59), since the difference between the two plans is not the binding of a variable but the fact that the action $\texttt{move}(a,b)$ is the final action in (61) (i.e., an immediate predecessor of the plan's terminal action) but not in (59). For similar reasons, a plan where two actions a_1 and a_2 are sequential is not of measure 0 in the plan where the actions are unordered.

Since (59) is of measure 0 in the plan set (60) and (61) is an instance (i.e., a subset) of (59), (61) is of measure 0 in (60) as well. Finally, (59) is also of measure 0 in the plan set given by removing

$$[\ldots\texttt{move}(a,c)\ldots] \tag{62}$$

from (60). After all, if binding ? to b reduces us to a set that is small relative to the set of all possible bindings, removing a single one of those possible bindings in advance shouldn't change this conclusion. The second condition in Definition 3.49 allows us to continue to measure the size of a plan set relative to (60) even after (62) has been removed.

What about adding new actions to a plan? If we add variable actions, the result will in general not be of measure 0 in the original plan; we are only committing to doing "something" and most of the linearizations of the original plan include additional actions of one form or another in any event. But if we add a specific action, the story is different:

Proposition 3.50 *Let P be a plan, and P' an instance of P with i the associated injection from A to A'. Then if there is any action in $A' - i(A)$ that is not a variable, P' is of measure 0 in P.*

Definition 3.51 *A plan set Q will be said to be* of measure 1 *in P if $P - Q$ is of measure 0 in P. Two plans sets will be called* approximately equal *if each is of measure 1 in the other.*

Lemma 3.52 *Q is of measure 0 in P if any of the following conditions holds:*

1. *Q is empty.*

2. *Q is a subset of a set of measure 0 in P.*

3. *Q is of measure 0 in a subset of P.*

4. *Q is of measure 0 in a superset S of P with P of measure 1 in S.*

Lemma 3.53 *Q is of measure 0 in P if and only if Q is of measure 0 in $P \cup Q$.*

Proposition 3.54 *Approximate equality is an equivalence relation.*

We also have the following:

Proposition 3.55 *Let $P \neq \emptyset$ be a plan set. Then provided that our language includes infinitely many object and action constants, there is no plan set that is both of measure 0 and of measure 1 in P.*

It is this result that gives teeth to the ideas we are proposing; if there were a plan of both measure 0 and measure 1 in P, we would be able to return as "generally correct" plans that in fact failed for large fractions of their linearizations.

The requirement that there be infinitely many constants is a necessary one. If, for example, there were only 37 places to which an object could be moved, we could use the fact that each specific choice is of measure 0 in the overall plan to conclude that the union of all of them was – thereby violating Proposition 3.55. Similarly, if the set of actions we could take were circumscribed in some way, we could use Proposition 3.50 to find a counterexample to the above proposition.

Finally, we present some technical results that we will need later. We begin by recalling the usual definition of convergence for a sequence S_i of sets:

Definition 3.56 *Let S_i be a sequence of (plan) sets. Then we will say that the S_i* converge *to a set S if for any x, there is some index $m(x)$ such that for $i > m(x)$, $x \in S_i$ if and only if $x \in S$.*

Lemma 3.57 *Suppose we have a sequence P_i of plan sets, where P_{i+1} is of measure 0 in P_i for each i. Then the P_i converge to the empty set.*

Every infinite descending chain where each element is of measure 0 in the previous one converges to the empty set.

It is *not* the case that there is no infinite descending chain of plan sets, each of measure 0 in the previous one. For any function constant f, we can get such a chain by considering

$$[\texttt{move}(x,?)] \supset [\texttt{move}(x,f(?))] \supset [\texttt{move}(x,f(f(?)))] \supset \cdots$$

Lemma 3.58 *Suppose S is of measure 1 in T and of measure 0 in U. Then T is of measure 0 in U.*

Suppose that we denote by $A \ominus B$ the symmetric difference of A and B, so that

$$A \ominus B = (A - B) \cup (B - A)$$

We now have:

Proposition 3.59 *Suppose that there is some set D that is of measure 1 in $A \ominus B$ and of measure 0 in A. Then A and B are approximately equal.*

3.3.3 Planning

Having introduced these notions, we need to use them to construct a planner. Before doing so, however, let us be clear about the problem that we are hoping to address. Our focus here is on planning itself, as opposed to reasoning about action or simulation. In other words, we will assume that the semantics of actions are provided to us; somewhat more specifically, we assume that given a linear plan L and a goal g, we have some way to tell whether or not g holds after L is executed. From a formal point of view, we will assume that given a goal g, we can take $L(g)$ to be the set of all linear plans that achieve g. The analysis we are about to present is independent of the specific semantics of action underlying the function L.

In the examples, of course, we will need to rely on a specific semantics of action. For the blocks world, this semantics is presumably intuitive and corresponds to the usual STRIPS description. The only difference between our interpretation and the conventional one is that we need some way to interpret actions that are attempted even though their preconditions are not satisfied; we will take the view that such actions simply have no effect on the domain in question.[29]

We now make the following definition:

Definition 3.60 *A planning system $\mathcal{P}$ accepts as input a goal g and a plan set P. It returns a plan set $\mathcal{P}(g,P) \subseteq P$ that is approximately equal to $L(g) \cap P$. In some cases, we will assume that the goal is fixed, writing $\mathcal{P}_g$ for the corresponding function that accepts the plan set P only.*

The plan set P can be used to focus the planner's attention on plans of a particular form. The condition that $\mathcal{P}(g,P)$ be of measure 1 in $L(g) \cap P$ means that almost all – but not necessarily all – of the plans in P that would achieve g are actually returned by the planner. In more picturesque terms, the planner is "approximately complete."

The condition that $L(g) \cap P$ be of measure 1 in $\mathcal{P}(g,P)$ means that almost all the plans returned by the planner achieve the goal; in other words, the planner is approximately correct. In a situation like this, where $L(g)$ is of measure 1 in a plan set P, we will often say that P "generally achieves" g.

In the remainder of this section, we will begin by discussing planning systems in general, describing implementation concerns that are likely to arise in their construction. There are then two technical issues that we will address. First, we will show that a planning system can be used to produce answers to planning queries that actually *are* correct and complete, at least in the limit. More precisely, we will show how a planning system can be used to construct a sequence of answers that converges on the actual set $L(g)$. Second, we will show how a planning system can respond to a conjunctive goal $g_1 \wedge g_2$ by invoking itself only on the subgoals g_1 and g_2 separately and then combining the results. More precisely, we will show how $\mathcal{P}_{g_1 \wedge g_2}$ can be constructed from $\mathcal{P}_{g_1}$ and $\mathcal{P}_{g_2}$.

We begin by discussing $\mathcal{P}_g$ itself. On an intuitive level, the way $\mathcal{P}_g$ works is as follows: Given the goal g, we find the actions a that might succeed in establishing g. If the precon-

[29] The axiomatization of Section 3.2.3 supports this.

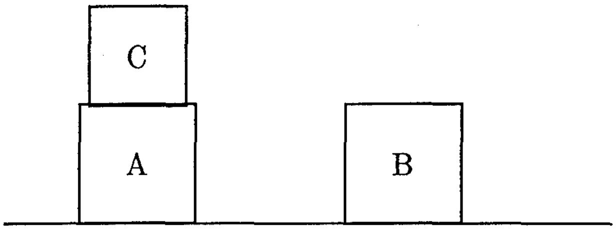


Figure 10: The Sussman anomaly

ditions to these actions are in general satisfied (perhaps because these preconditions hold in the initial situation), we can take $\mathcal{P}(G)$ to be the union of plan sets of the form

$$[\ldots a \ldots] \tag{63}$$

for each action a achieving g.

If the preconditions of a are not in general satisfied, we can invoke $\mathcal{P}$ recursively on each precondition, merge the results to obtain plans that enable a, and then append a to the end of such plans. The result (63) is in fact a special case of this observation; if the preconditions to a are known to hold in the initial state [ ], these preconditions also hold in a plan set that is approximately equal to [...] (the set of all plans). The expression (63) is simply the result of appending a to the end of such plans.

We will see in what follows that we are often interested not only in $\mathcal{P}_g$, which constructs plans for achieving g, but also in $\mathcal{P}_{\neg g}$, which tells us which elements of a particular plan set *fail* to achieve g. This has obvious analogs in existing planners:

1. In a system like TWEAK [8], the exceptions involve finding what Chapman calls *clobberers.* New actions that make the plan work after all are called *white knights.*

2. McAllester and Rosenblitt [65] describe potential flaws in a plan (where one action might overturn the consequences of another) as *threats.* Overcoming the threats involves adding new actions to the plan that ensure that the consequences of the action hold after all.

3. Finally, we will discuss in Section 3.3.4 the use of our declarative axiomatization of action to construct $\mathcal{P}_g$ and $\mathcal{P}_{\neg g}$.

Before turning to technical issues, let us look at an example in a bit more detail. The problem we will consider is the well-known Sussman anomaly [101], shown in Figure 10. The goal is to get A on B and B on C. At this point, we consider the subgoals separately.

The first of these involves simply getting B on C. This is generally achieved by the plan

$$[\ldots \texttt{move}(B, C) \ldots] \tag{64}$$

Although there are instances of (64) that do not succeed in getting B on C, there are only a finite number of ways for this to happen – something must be put on B or on C, or B has to be moved away from C at the end of the plan. Each of these exceptions is of measure 0 in (64), which is why the plan (64) generally achieves on(B, C). Furthermore, move(B, C) is the only action with on(B, C) in its add list, and the preconditions to this action hold in the initial situation. This implies that the plan set of (64) is approximately equal to the set of all plans for getting B onto C and we can take

$$\mathcal{P}(\text{on}(B,C),[\ldots]) = [\ldots\texttt{move}(B,C)\ldots] \tag{65}$$

The two conditions of approximate equality are satisfied: Most plans that achieve the goal are instances of the above plan (in fact, they all are), and the exceptions are a set of measure 0 in (65).

To continue the analysis, we compute

$$\mathcal{P}(\neg\text{on}(B,C),[\ldots\texttt{move}(B,C)\ldots])$$

in order to determine which elements of (65) are exceptions to the plan. There are two ways in which such exceptions might arise: One of the preconditions to the move action might fail to hold, or something might clobber the fact that B is on C after the action is executed.

The action move(B, C) has two preconditions, that B be clear and that C be. B is clear in the initial situation, and the most general plan that clobbers this fact is

$$[\ldots\texttt{move}(?,B)\ldots]$$

It follows that the plan (64) will fail for the instance

$$[\ldots\texttt{move}(?,B)\ldots\texttt{move}(B,C)\ldots] \tag{66}$$

There are still more specific plans that do manage to get B on C, but (66) is one general failure possibility. (Recall that once we move something onto B, we are assuming that the failed action of moving B to C simply has no effect on the locations of the blocks.) Another way for (64) to fail is given by

$$[\ldots\texttt{move}(?,C)\ldots\texttt{move}(B,C)\ldots] \tag{67}$$

where something is moved onto the top of C.

The only remaining possibility is where B is not on C at the end of the plan because it is moved away. Combining this with (66) and (67), we see that we can take

$$\mathcal{P}(\neg\text{on}(B,C),[\ldots\texttt{move}(B,C)\ldots]) = e_1 \cup e_2 \cup e_3$$

to be the union of the following three sets of exceptions:

$$\begin{aligned} e_1 &= [\ldots\texttt{move}(?,B)\ldots\texttt{move}(B,C)\ldots] \\ e_2 &= [\ldots\texttt{move}(?,C)\ldots\texttt{move}(B,C)\ldots] \\ e_3 &= [\ldots\texttt{move}(B,C)\ldots\texttt{move}(B,?)\ldots] \end{aligned} \tag{68}$$

Each of these sets is of measure 0 in (65), as is their union.

If we wish, we can continue the process, computing

$$\mathcal{P}(\mathrm{on}(B,C), e_1 \cup e_2 \cup e_3)$$

to find those elements of the exception set that achieve the goal after all, and so on. As we will see shortly, a sequence constructed in this fashion will eventually converge on the set of all plans that achieve the goal of getting B on C.

The second goal $\mathrm{on}(A,B)$ is more complicated, but only slightly so. It is generally achieved by

$$[\ldots\mathtt{move}(C,?)\ldots\mathtt{move}(A,B)\ldots] \tag{69}$$

Once again, there are only finitely many ways for (69) to fail – the binding for ? could be chosen poorly (A and B are bad choices), or additional actions could be added as in the previous case. (In keeping with Proposition 3.55, we are assuming that there are an infinite number of distinct locations on the table to which C could be moved.) The complete list of exceptions is as follows:

$$\begin{aligned}
f_1 &= [\ldots\mathtt{move}(?1,C)\ldots\mathtt{move}(C,?)\ldots\mathtt{move}(A,B)\ldots]\\
f_2 &= [\ldots\mathtt{move}(?1,?)\ldots\mathtt{move}(C,?)\ldots\mathtt{move}(A,B)\ldots]\\
f_3 &= [\ldots\mathtt{move}(C,?)\ldots\mathtt{move}(?1,A)\ldots\mathtt{move}(A,B)\ldots]\\
f_4 &= [\ldots[\mathtt{move}(?1,B)\ \&\ \mathtt{move}(C,?)]\ldots\mathtt{move}(A,B)\ldots]\\
f_5 &= [\ldots\mathtt{move}(C,?)\ldots\mathtt{move}(A,B)\ldots\mathtt{move}(A,?1)\ldots]\\
f_6 &= [\ldots\mathtt{move}(C,A)\ldots\mathtt{move}(A,B)\ldots]\\
f_7 &= [\ldots\mathtt{move}(C,B)\ldots\mathtt{move}(A,B)\ldots]
\end{aligned} \tag{70}$$

In f_4, we have extended our notation somewhat, writing a & b for the plan of taking actions a and b without there being an ordering constraint between them.

In less formal terms, the plan (69) can fail for the following reasons:

1. The attempt to move C out of the way can fail. This may happen because something has been moved on top of C (f_1) or because something has already been moved to C's intended destination (f_2).

2. Something may be moved onto A (f_3 or f_6) or onto B (f_4 or f_7). We get two exceptions in each case here depending on whether the block moved into the way is C (f_6 and f_7) or not (f_3 and f_4).

3. A may be moved off of B after it is put there (f_5).

Because all of the exceptions are of measure 0 in (69), (69) itself is a satisfactory choice for $\mathcal{P}(\mathrm{on}(A,B),[\ldots])$. We also take

$$\mathcal{P}(\neg\mathrm{on}(A,B),[\ldots\mathtt{move}(C,?)\ldots\mathtt{move}(A,B)\ldots]) = \cup_i f_i$$

We needed to be careful when constructing the above exceptions to take f_3 as indicated instead of the more obvious choice

$$f_3' = [\ldots[\mathtt{move}(?1, A)\ \&\ \mathtt{move}(C, ?)]\ldots\mathtt{move}(A, B)\ldots] \tag{71}$$

where we have allowed the action of moving something onto A to occur in parallel with the action of moving C out of the way. The reason is that if the action of moving ?1 onto A is to interfere with the rest of the plan, this action must succeed – and it won't unless C is moved out of the way first. Put more formally, the subset of f_3' that actually achieves on(A, B) includes a component that is approximately equal to

$$[\ldots\mathtt{move}(?1, A)\ldots\mathtt{move}(C, ?)\ldots\mathtt{move}(A, B)\ldots] \tag{72}$$

and is therefore not of measure 0 in f_3'. Again, recall that our semantics of failed actions is that they have no effect at all. Since A is not clear in Figure 10, the first action of (72) fails and (72) effectively reduces to (69).

From a computational point of view, it may be more attractive to work with f_3' than to work with the more accurate f_3 appearing in (70). The reason is that f_3' is already of measure 0 in the original plan (69), and it is simpler than f_3 and therefore presumably easier to generate. It is obviously easier to stop as soon as a set of measure 0 in the original plan is encountered than to complete the analysis to obtain f_3 instead of f_3'. This is exactly what commonsense planners should do – when we plan for one of a set of conjuncts, we only worry about what might go wrong until we feel confident in dismissing it. In our running example, we know that something will go wrong with the plan of moving A to B if we move an additional block onto A. The need for this extra action ensures that we are looking at a set of measure 0 in our overall plan, so we don't think about it further. More specifically, we don't bother to draw the conclusion that we can only move something onto A after C is cleared off the top of it. It is to remain in keeping with this approach that may wish to work with f_3' instead of f_3.

The general version of this construction is similar. At each odd-numbered step (including the first), we look for plans that achieve the goal but that we have not yet identified. At even-numbered steps, we look for exceptions to the plans found thus far:

Definition 3.61 *Given a planning system $\mathcal{P}$ and a goal g, the* planning sequence generated by $\mathcal{P}$ for g *is given by*

$$\mathcal{P}_i(g) = \begin{cases} \mathcal{P}_{i-1}(g) \cup \mathcal{P}(g, P - \mathcal{P}_{i-1}(g)), & \text{if } i \text{ is odd;} \\ \mathcal{P}_{i-1}(g) - \mathcal{P}(\neg g, \mathcal{P}_{i-1}(g)), & \text{if } i \text{ is even.} \end{cases} \tag{73}$$

The sequence is initialized by $\mathcal{P}_0(g) = \emptyset$.

Some notation will make this definition easier to work with. If we write $\mathcal{D}_i$ for the symmetric difference between $\mathcal{P}_i$ and $\mathcal{P}_{i-1}$, it suffices to describe only how $\mathcal{D}_i$ is computed at each step. We know that $\mathcal{D}_i$ has to be added to $\mathcal{P}_i$ at odd steps and removed at even steps. Now (73) becomes:

$$\mathcal{D}_i(g) = \begin{cases} \mathcal{P}(g, P - \mathcal{P}_{i-1}(g)), & \text{if } i \text{ is odd;} \\ \mathcal{P}(\neg g, \mathcal{P}_{i-1}(g)), & \text{if } i \text{ is even.} \end{cases} \tag{74}$$

Further simplification is often possible as well. If i is even, for example, we can evaluate (74) recursively to get

$$\mathcal{D}_i(g) = \mathcal{P}(\neg g, \mathcal{P}_{i-2}(g) \cup \mathcal{D}_{i-1}(g)) \tag{75}$$

If we know that we caught all of the exceptions at the $i-$ 2nd step, we will know that $\mathcal{P}(\neg g, \mathcal{P}_{i-2}(g)) = \emptyset$ and we can replace (75) with the simpler

$$\mathcal{D}_i(g) = \mathcal{P}(\neg g, \mathcal{D}_{i-1}(g)) \tag{76}$$

In a similar way, if we know that $\mathcal{D}_{i-1}(g)$ includes all the plans that achieve g at an odd step, we can conclude

$$\mathcal{D}_i(g) = \mathcal{P}(g, \mathcal{D}_{i-1}(g)) \tag{77}$$

In both (76) and (77), the purpose of each step is to correct possible incorrectness in the previous step; possible incompleteness in the previous step is not an issue.

We are now in a position to achieve the first of our two technical goals:

Theorem 3.62 *Given a planning system $\mathcal{P}$ and a goal g, the planning sequence generated by $\mathcal{P}$ for g converges to $L(g)$.*

This result shows us how to construct a planner that is correct and complete from one that is approximately correct and approximately complete.

Our remaining goal is that of showing how to combine the planner's results for g_1 and for g_2 to obtain a plan for $g_1 \wedge g_2$. Presumably, the semantics underlying L is such that a plan achieves the conjunctive goal $g_1 \wedge g_2$ if and only if it achieves both g_1 and g_2, so that

$$L(g_1 \wedge g_2) = L(g_1) \cap L(g_2)$$

The following result is now obvious:

Lemma 3.63 *Suppose that P_1 achieves a goal g_1 and that P_2 achieves g_2. Then $P_1 \cap P_2$ achieves $g_1 \wedge g_2$.*

The problem, of course, is that $P_1 \cap P_2$ may well be empty if P_1 and P_2 are specific linear plans that achieve the goals. If we could weaken the above lemma to require only that the plans *generally* achieve their goals, we could use the fact that (64) generally achieves on(B, C) and that (69) generally achieves on(A, B) to conclude that a general solution to the Sussman anomaly is

$$[\ldots\texttt{move}(B, C)\ \&\ [\texttt{move}(C, ?)\ldots\texttt{move}(A, B)]\ldots] \tag{78}$$

This plan involves three actions – moving B to C, moving C out of the way, and moving A to B. C must be moved before A is put on B, but there are no other ordering constraints involved. Unfortunately, this is not a solution to the Sussman anomaly, since it allows the action of moving B to C to precede the action of moving C out of the way. Here is the general problem:

Proposition 3.64 *There exist plans P_1, P_2, Q_1 and Q_2 with Q_i of measure 0 in P_i but $Q_1 \cap Q_2$ of measure 1 in $P_1 \cap P_2$.*

The Sussman anomaly isn't quite this bad; the correct plan

$$[\ldots\texttt{move}(C,?)\ldots\texttt{move}(B,C)\ldots\texttt{move}(A,B)\ldots]$$

is neither of measure 0 nor of measure 1 in (78). If it were of measure 1 in (78), then (78) would generally achieve the original goal of getting A on B and B on C; if it were of measure 0, (78) would in general fail to achieve the goal. In fact, it succeeds some of the time and fails in others; it depends only on how we order the actions.

The tower-construction problem of Figure 9 is an example where Proposition 3.64 does hold. The plan

$$[\ldots\texttt{move}(B,C)\ldots]$$

gets B on C, and

$$[\ldots\texttt{move}(A,B)\ldots]$$

generally gets A on B. Nevertheless the plan for constructing the tower is of measure 0 in

$$[\ldots\texttt{move}(A,B)\ \&\ \texttt{move}(B,C)\ldots]$$

because we must take the additional action of moving C to the table. In terms of the proposition, the P_i are the plans for achieving the subgoals, and the Q_i are the exception sets for these plans.

We cannot necessarily merge specific plans for achieving the individual conjuncts. Nor, as Proposition 3.64 tells us, can we necessarily find a plan for the conjunctive goal by merging plans for *generally* achieving the conjuncts. But we do have the following, an immediate consequence of Theorem 3.62:

Corollary 3.65 *Given a planning system $\mathcal{P}$ and goals g_1 and g_2, denote by $\mathcal{P}_i(g)$ the planning sequence generated by $\mathcal{P}$ for g. Then the sequence*

$$\mathcal{P}_i(g_1) \cap \mathcal{P}_i(g_2) \tag{79}$$

converges to $L(g_1 \wedge g_2)$, the plan for achieving the conjunction of g_1 and g_2.

This result is evidence that we are on the right track, although we need to do a bit better – taking the limit in (79) will not be viable in practice. More precisely, we need a way to compute $\mathcal{P}_{g_1 \wedge g_2}$ that we can guarantee to satisfy the requirements of Definition 3.60. To see how to do this, let us look at the Sussman anomaly in a bit more detail.

We already know how to construct the first two terms in the planning sequences for the subgoals; they are

$$\begin{aligned}
\mathcal{P}_1(\mathrm{on}(A,B)) &= [\ldots \mathtt{move}(C,?) \ldots \mathtt{move}(A,B) \ldots] \\
\mathcal{P}_2(\mathrm{on}(A,B)) &= \mathcal{P}_1(\mathrm{on}(A,B)) - \cup f_i \\
\mathcal{P}_1(\mathrm{on}(B,C)) &= [\ldots \mathtt{move}(B,C) \ldots] \\
\mathcal{P}_2(\mathrm{on}(B,C)) &= \mathcal{P}_1(\mathrm{on}(B,C)) - \cup e_i
\end{aligned}$$

where the e_i and f_i are given by (68) and (70) respectively.

Let us denote the conjoined sequence in (79) by simply $\mathcal{P}_i$. It now follows that the first element of this sequence is given by

$$\mathcal{P}_1 = [\ldots \mathtt{move}(B,C) \; \& \; [\mathtt{move}(C,?) \ldots \mathtt{move}(A,B)] \ldots] \tag{80}$$

The second element of the sequence involves removing from $\mathcal{P}_1$ the exceptions in either (68) or (70). We can compute these by combining, for example, the plan

$$e_1 = [\ldots \mathtt{move}(?1,B) \ldots \mathtt{move}(B,C) \ldots] \tag{81}$$

which is one of the three sets removed from $\mathcal{P}_1(\mathrm{on}(B,C))$, with

$$[\ldots \mathtt{move}(C,?) \ldots \mathtt{move}(A,B) \ldots] \tag{82}$$

which is the original $\mathcal{P}_1(\mathrm{on}(A,B))$. (We have standardized apart the variables in e_1 and $\mathcal{P}_1(\mathrm{on}(A,B))$, which is why the ? in (68) has been replaced with ?1 in (81).) The result of this particular merge is the following set of three plans:

$$[\ldots [\mathtt{move}(?1,B) \ldots \mathtt{move}(B,C)] \; \& \; [\mathtt{move}(C,?) \ldots \mathtt{move}(A,B)] \ldots] \tag{83}$$

$$[\ldots \mathtt{move}(C,?) \ldots \mathtt{move}(A,B) \ldots \mathtt{move}(B,C) \ldots] \tag{84}$$

$$[\ldots \mathtt{move}(C,B) \ldots [\mathtt{move}(B,C) \; \& \; \mathtt{move}(A,B)] \ldots] \tag{85}$$

The first of the above plans is the "obvious" merge where the two separate plans are simply executed in parallel. In the second, the variable ?1 is bound to A and the first action in e_1 is identified with the second action in $\mathcal{P}_1(\mathrm{on}(A,B))$. The ordering on the resulting action sequence is accumulated from the orderings on e_1 and on $\mathcal{P}_1(\mathrm{on}(A,B))$. The third plan is similar, with ? being bound to B and ?1 bound to C.

Each of the three plans fails to achieve the subgoal of getting B onto C. In (83), a new (and currently unidentified) block is moved onto B. In (84), A is moved onto B before B is moved onto C. And finally, C itself is moved onto B in (85).

It is only (84) that is of interest to us. The plan (83) is of measure 0 in the set of exceptions because it involves an additional action, and (85) is of measure 0 because it binds the variable ?. As we have already remarked, (84) tells us that if we move A to B before moving B to C, we will not achieve our overall goal because B will be occupied when we try to move it.

We can continue in this fashion, accumulating all of the exceptions to the overall plan (80). In addition to (84), the only plan not of measure 0 in the set of all exceptions is

$$[\ldots\texttt{move}(B,C)\ldots\texttt{move}(C,?)\ldots\texttt{move}(A,B)\ldots]$$

which is part of the result of merging $\mathcal{P}_1(\text{on}(B,C))$ and f_1; this tells us that if we move B to C too early, our plan for getting C out of the way en route to moving A will fail.

We can conclude from all this that a generally valid plan for solving the Sussman anomaly is given by removing from the plan (80) the union of the two plans

$$[\ldots\texttt{move}(C,?)\ldots\texttt{move}(A,B)\ldots\texttt{move}(B,C)\ldots]$$
$$[\ldots\texttt{move}(B,C)\ldots\texttt{move}(C,?)\ldots\texttt{move}(A,B)\ldots]$$

The result is equivalent to the plan

$$[\ldots\texttt{move}(C,?)\ldots\texttt{move}(B,C)\ldots\texttt{move}(A,B)\ldots] \tag{86}$$

which is indeed the usual solution to the original problem.

Here is the result dealing with the general situation:

Theorem 3.66 *Suppose that we have a conjunctive goal $g_1 \wedge g_2$ and a set P. Construct the plan sequences P_i converging to $L(g_1) \cap P$ and Q_i converging to $L(g_2) \cap P$. Now we can always find an i and a j such that both $P_i \ominus P_{i+1}$ and $Q_j \ominus Q_{j+1}$ are of measure 0 in $P_i \cap Q_j$. For any such i and j, $P_i \cap Q_j$ will be approximately equal to $L(g_1 \wedge g_2) \cap P$.*

In our analysis of the Sussman anomaly, we actually terminated the construction of the plans for the subgoals somewhat earlier than the points sanctioned by the above result. This early termination reflects some lookahead on our part; consider, for example, the fact that the exception

$$e_1 = [\ldots\texttt{move}(?,B)\ldots\texttt{move}(B,C)\ldots] \tag{87}$$

to the plan for getting B on C, when combined with the plan

$$[\ldots\texttt{move}(C,?)\ldots\texttt{move}(A,B)\ldots] \tag{88}$$

for getting A onto B, led to the exception

$$[\ldots\texttt{move}(C,?)\ldots\texttt{move}(A,B)\ldots\texttt{move}(B,C)\ldots] \tag{89}$$

to the general plan of moving B onto C in combination with (88).

It turns out, however, that one set of measure 0 in e_1 that actually achieves on(B,C) is

$$[\ldots\texttt{move}(A,B)\ldots\texttt{move}(B,C)\ldots] \tag{90}$$

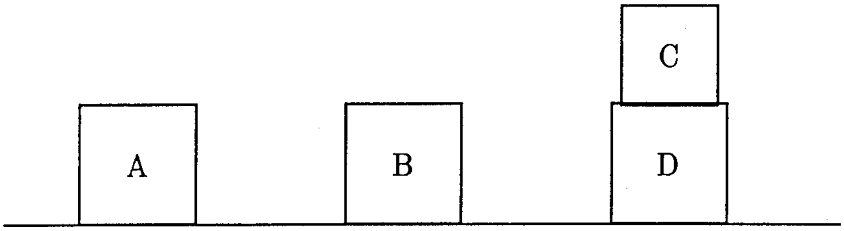


Figure 11: Get A on B on C without building a 4-block tower

The reason for this is that the initial action of moving A to B will fail (C is still in the way), so B will wind up on C after all. Now (89) is an instance of (90) and therefore might *not* be an exception to the general plan of getting B onto C.

The recognition that binding ? to A in $[\ldots\texttt{move}(?,B)\ldots\texttt{move}(B,C)\ldots]$ is an exception to the overall plan is subtle. Roughly speaking, we need the action of moving A to B to succeed (in order to achieve the other subgoal), so moving B to C will indeed be blocked. In terms of Theorem 3.66, (90) isn't an exception to the plan for getting B on C, but

$$[\ldots\texttt{move}(C,?)\ldots\texttt{move}(A,B)\ldots\texttt{move}(B,C)\ldots] \tag{91}$$

is an exception, and that's what matters. Once we have identified (91) as an exception to the original plan, the conditions of Theorem 3.66 are satisfied and we can construct the overall plan (86) with confidence.

As a final example, let us consider the tower-construction problem once again. The problem is repeated in Figure 11; recall that the goal is to get A on B and B on C without ever building a 4-block tower.

The planning sequence for getting B on C begins with

$$P_1 = [\ldots\texttt{move}(B,C)\ldots]$$

and exceptions given by

$$\begin{aligned}
e_1 &= [\ldots\texttt{move}(?,B)\ldots\texttt{move}(B,C)\ldots] && (92)\\
e_2 &= [\ldots\texttt{move}(?,C)\ldots\texttt{move}(B,C)\ldots] &&\\
e_3 &= [\ldots\texttt{move}(B,C)\ldots\texttt{move}(B,?)\ldots] &&\\
e_4 &= [\ldots\texttt{move}(?,?1)\ldots\texttt{move}(C,?)\ldots\texttt{move}(B,C)\ldots] && (93)
\end{aligned}$$

The final exception above involves situations where B cannot be moved to C because a 4-block tower is involved. In practice, however, the exception (93) is less likely to be generated than is

$$e_4' = [\ldots\texttt{move}(C,?)\ldots\texttt{move}(B,C)\ldots] \tag{94}$$

We realize that putting B on C can never violate the 4-block constraint unless we first move C *somewhere*. We will temporarily work with e_4' instead of e_4, just as we indicated the possibility of working with f_3' in (71) instead of f_3 in (70) in the Sussman anomaly.

In a similar way, the planning sequence for getting A on B begins with

$$Q_1 = [\ldots\texttt{move}(A,B)\ldots] \tag{95}$$

and exceptions

$$\begin{aligned}
f_1 &= [\ldots\texttt{move}(?,A)\ldots\texttt{move}(A,B)\ldots] \\
f_2 &= [\ldots\texttt{move}(?,B)\ldots\texttt{move}(A,B)\ldots] \\
f_3 &= [\ldots\texttt{move}(A,B)\ldots\texttt{move}(A,?)\ldots] \\
f_4' &= [\ldots\texttt{move}(B,?)\ldots\texttt{move}(A,B)\ldots] \qquad (96)
\end{aligned}$$

When we combine this sequence with the previous one, we get

$$P_1 \cap Q_1 = [\ldots\texttt{move}(A,B)\ \&\ \texttt{move}(B,C)\ldots]$$

and the exceptions include

$$\begin{aligned}
g_1 &= [\ldots\texttt{move}(A,B)\ldots\texttt{move}(B,C)\ldots] \qquad (97)\\
g_2 &= [\ldots\texttt{move}(B,C)\ldots\texttt{move}(A,B)\ldots] \qquad (98)
\end{aligned}$$

together with sets of measure 0 with respect to these. But

$$g_1 \cup g_2 = [\ldots\texttt{move}(A,B)\ \&\ \texttt{move}(B,C)\ldots]$$

since both possible orderings are eliminated. From a commonsense point of view, we can't put A on B first because we will then be unable to get B to C, and can't put B on C first because this might (and in fact does) make a 3-block tower to which A cannot be added.

There are two ways in which the analysis can be extended. The exceptions (97) and (98) are the result of intersections with (92) and (96), so one of these two sets must be analyzed further. Since f_4' is an approximation, it seems natural to work on this first, leading to

$$\begin{aligned}
f_1 &= [\ldots\texttt{move}(?,A)\ldots\texttt{move}(A,B)\ldots] \\
f_2 &= [\ldots\texttt{move}(?,B)\ldots\texttt{move}(A,B)\ldots] \\
f_3 &= [\ldots\texttt{move}(A,B)\ldots\texttt{move}(A,?)\ldots] \\
f_4 &= [\ldots\texttt{move}(B,C)\ldots\texttt{move}(A,B)\ldots] \qquad (99)\\
f_5 &= [\ldots\texttt{move}(?,?1)\ldots\texttt{move}(B,?)\ldots\texttt{move}(A,B)\ldots] \qquad (100)
\end{aligned}$$

The final line (100) indicates that one way to make B the top block in a 3-block stack is to move `?` to `?1` and then B to `?`. This is of measure 0 in our prospective solution (95), however, so we need not worry about it.[30] The problem continues to be (99), which is still enough to invalidate our original plan. We now have to choose between finding exceptions to

[30] As in the Sussman anomaly, we actually need to be a bit more careful but the details are not of interest.

(99), finding a way to move B to C without creating a 3-block tower, and finding exceptions to (92), finding a way to move A to B before moving B to C.

Let us suppose that we choose wrongly, so that we now have to look for instances of

$$[\ldots\texttt{move}(?,B)\ldots\texttt{move}(B,C)\ldots]$$

for which B actually ends up on C after all. Here is the most general solution:

$$[\ldots\texttt{move}(?,B)\ldots\texttt{move}(?,?1)\ldots\texttt{move}(B,C)\ldots] \tag{101}$$

Unfortunately, this doesn't help us with our original difficulty, since we need to bind ? to A and now

$$[\ldots\texttt{move}(A,B)\ldots\texttt{move}(A,?1)\ldots] \tag{102}$$

is known to be an exception that fails to get A onto B.

So we turn our attention to (99); if we begin by moving C, then we will in fact be able to move A to B after all. So we can achieve on(A,B) using the plan

$$[\ldots\texttt{move}(C,?)\ldots\texttt{move}(B,C)\ldots\texttt{move}(A,B)\ldots] \tag{103}$$

Unfortunately, this still might not work, since moving C (potentially to the top of a 3-block stack) may cause a problem in getting B to C as indicated in the original plan (94) for achieving this subgoal. But now we finally replace (94) with the more appropriate (93), allowing us to conclude that (103) does indeed generally achieve the goal of getting A on B and B on C.

The analysis would be very different if we were to work with a more conventional planner. There, the fact that we cannot build a 4-block tower would be encoded by adding a new precondition to the move operator, saying that in order to move x to y, either y must be on the table, or it must be on a block that is on the table.

Now when we try to move B to C, we will naturally generate the plan of first moving C to the table (since C being on the table is one possible way to achieve the disjunctive precondition). This plan can then be extended to solve the problem but the plan itself has been constructed blindly instead of in response to an identified bug in the simple plan of putting B on C and then A on B.

In our approach, the problem is identified in (96), which is later refined into (99) and (103). A plan to overcome the 4-block difficulty is generated only when it is needed, and not as part of a general attempt to get B onto C.

There is another difference in the treatments of this example as well. Consider a conventional planner that proceeds by first planning to move B to C, and then planning to move A to B. When the difficulty is found and the plan for getting B onto C has to be modified, the ensuing backtrack will discard the plan for getting A onto B. Not much work is lost in our simple example, but in more complex problems it may be crucial to avoid replanning for the goal of getting A onto B. After all, the existing plan for achieving this subgoal is the correct one.

The approach that we have described behaves in this fashiòn. The successive refinements to the plan for getting A onto B, beginning with thc basic plan

$$[\ldots\mathtt{move}(A,B)\ldots] \tag{104}$$

and eventually culminating in

$$[\ldots\mathtt{move}(C,?)\ldots\mathtt{move}(B,C)\ldots\mathtt{move}(A,B)\ldots]$$

all continue to use the fundamental plan (104). In fact, work is in some sense *never* discarded in our approach, since we proceed by gradually refining plan sets in ways that are suggested by the merging computations and by corresponding interactions among subgoals. In our framework, the plan for constructing the tower is built up by starting with the basic plans of moving B to C and A to B, and then debugging the result. This leads to a much more focussed search process than that associated with conventional methods.

3.3.4 Implementation considerations

In order to actually build a planning system based on the ideas that we have described, there are three separate problems that need to be addressed. First, we need to discuss the manipulation of plan sets, including the underlying operation of plan intersection. Second, we need to describe the construction of a system that can produce the plan sets in the first place. And finally, we need to discuss implementation details surrounding results like Theorem 3.66; there are several simple ideas that can make this result substantially more effective in practice. (Witness the footnote in the previous section.) We will deal with these issues in this order.

Plan intersection and manipulating plan sets Plan intersection is often known as plan *merging* and has already been discussed by a variety of authors; typical is the treatment of Foulser *et.al* [28]. The construction described there is related to ours, although not identical. Foulser *et.al* allow for the possibility that more than two plans be merged at once, but their plan description language is more restricted than ours. In keeping with conventional interests, they assume that the actions in a plan are sequential; no others can be interspersed as new information is obtained. As a result, they do not draw the distinction between $\leq$ and $*$ that was our focus in Section 3.3.1.

Nevertheless, the ideas introduced by Foulser and his coauthors [28] can be used to implement our somewhat more general notion of plan intersection. The method used continues to be that of treating sequences of actions (actions related by $\bar{*}$ in our notation) as atomic units, and then merging larger structures made up of these units. The details are not of any great theoretical interest and the interested reader is referred to the code itself.[31]

One thing that does bear mention is that the result of intersecting two plans may be a plan set that cannot be represented as a single plan; witness the construction of (83), (84)

[31] The code described in this section is part of the MVL theorem pṛoving system [33, 37, 41].

and (85) from the intersection of (81) and (82). The implementation obviously needs to cater to this possibility.

Manipulating general plan sets is a bit more interesting. This is quite a difficult problem but is made simpler in practice by the recognition that the plan sets under consideration are generally of the form

$$D_1 - D_2 \cup D_3 - D_4 \cup D_5 - \cdots \tag{105}$$

where the D_i are the symmetric differences of successive $\mathcal{P}_i$ and are in general fairly simply represented. The evaluation here is intended to be from left to right, so that (105) is in fact

$$(((D_1 - D_2) \cup D_3) - D_4) \cup D_5 - \cdots$$

The implementation is constructed in just this way, representing any particular plan set as an alternating sum such as (105).[32] Taking the union or intersection of these alternating sums is tedious but not terribly difficult.

Of course, the manipulations involved are fundamentally dependent on the plan intersection operation that we described earlier. Existing planners work with global data structures, gradually accumulating actions into an overall plan. This makes their plans somewhat brittle, since they will need to discard a great deal of existing work when a portion of the plan changes. A system such as we have described plans for subgoals separately but must frequently merge the plans involved in order to understand possible subgoal interactions.

The upshot of this is that the speed of a planner built on our ideas will be crucially dependent on the speed of the underlying mechanism for plan merging; as Foulser *et.al* point out, plan merging can be exponentially expensive. They also point out, however, that this exponential expense appears not to be incurred in practice because the plans being merged consist of small numbers of linear segments. This matches our experience. Finally, since specific plan segments tend to appear many times in the analysis, the overall computation can be speeded substantially by caching the results of the merging computation in some way.[33]

Constructing plan sets Given an implementation that manipulates plan sets, from where are we to obtain these sets in the first place? There are three sources:

1. Information about the initial situation (corresponding to the plan [ ]) can be encoded in this fashion. Thus in the tower-construction example, we know that $\text{on}(C, D)$ is definitely achieved by the plan [ ]. It follows that $\text{on}(C, D)$ is generally achieved by the universal plan set [...].

2. Information about actions occurring (although perhaps not succeeding) is encoded similarly. For any action a, the statement "a has just occurred" is true for the plan $[\ldots a]$. What this says that that a occurs at the end of any sequence of actions that does indeed end in a.

[32] The description of (105) continues to satisfy the criterion of being *event-driven* (see Section 3.2), since it records information only about those points where the value of a function from plan sets to B changes.

[33] More effective still appears to be to cache the results of the plan *instance* computation.

3. Finally, there are operations that transform plan sets into new plan sets. We have already seen one of these in the form of intersection; another corresponds to the frame axiom in our setting.

Definition 3.67 *There exists a* frame operator $\mathcal{F}$ *that accepts as input two plan sets and returns another plan set. The operator is defined so that if a particular goal or fluent g is inserted into the database by those plans in the set $\mathcal{P}^+$ and deleted from the database by those plans in $\mathcal{P}^-$, then $L(g) = \mathcal{F}(\mathcal{P}^+, \mathcal{P}^-)$.*

Somewhat less formally, the goal is achieved by plans in $\mathcal{F}(\mathcal{P}^+, \mathcal{P}^-)$ and not achieved by plans outside this set. Note that the equality in the definition is exact, not approximate.

As an example, we would expect to have

$$\mathcal{F}([\,], \emptyset) = [\ldots] \tag{106}$$

What this tells us is that if a fluent is true in the initial situation, and we know of no reason for it to be false in other situations, we can expect it to be true at all times.

Here is another example:

$$\mathcal{F}([\ldots a], \emptyset) = [\ldots a \ldots]$$

If an action a succeeds in achieving a goal and no other actions delete that goal, then we can use the frame axiom to conclude that the goal continues to hold after additional actions occur.

Here is a slightly more interesting example. Suppose that some goal is added to the database by the plan $[\ldots a]$ but deleted by $[\ldots b]$. What should $\mathcal{F}([\ldots a], [\ldots b])$ be?

The result should be

$$[\ldots a \ldots] - [\ldots b \ldots] \cup [\ldots b \ldots a \ldots] \tag{107}$$

since if both actions a and b occur, the goal will be true only if a occurs after b does. This fits neatly into the implementation details already discussed; the plan set (107) is conveniently written as an alternating sum.

It is because of examples such as this one that the operator $\mathcal{F}$ is defined on *pairs* of plan sets. We cannot find a unary $\mathcal{F}'$ such that

$$\mathcal{F}(P, Q) = \mathcal{F}'(P) - \mathcal{F}'(Q)$$

because, as we see from (107), the plan sets P and Q interact in the construction of $\mathcal{F}(P, Q)$.

The frame operator underlies the planning version of the modal operator propagate that we discussed in Section 3.2.2. The bilattice in question takes as its temporal elements not the integers, but the set of all linear plans; suppose for a moment that the base bilattice B is the first-order bilattice F. For a functional truth value g, $g \vee u$ corresponds to the points at which g is true and $\neg g \wedge u$ to the points at which g is false. For a specific plan P, we therefore define

$$\texttt{propagate}(g)(P) = \begin{cases} t, & \text{if } P \in \mathcal{F}(g \vee u, \neg g \wedge u); \\ f, & \text{otherwise.} \end{cases} \tag{108}$$

Generalizations of (108) that handle other base bilattices are straightforward.

We can also define **delay** in this setting. If P is a linear plan, we will denote by $\langle P|a\rangle$ the result of appending the action a to the end of P. The planning version of (47) is now simply:

$$[\mathtt{delay}(f)]\langle P|a\rangle = f(P) \tag{109}$$

The final modifications needed to reduce planning to theorem proving in a bilattice setting are the introduction of a specific axiom

$$\mathtt{occurs}(a)$$

indicating that the action a occurs at the end of any linear plan that is an instance of the plan set $[\ldots a]$, and another axiom to the effect that occurs(init) is true for the plan set $[\,]$.

Status The current implementation handles plans and plan sets as described. The examples of Section 3.3.3 make some specific control assumptions about the nature of the search, and these control decisions are not yet supported by the implementation. (In a bilattice setting, they appear to be restrictions to planning of more general control heuristics, and we are attempting to implement these general control notions as opposed to specializations of them.)

Rather than invoke Theorem 3.66 directly, the planner works by determining at each point whether a particular line of reasoning will impact its overall answer. In other words, it decides whether or not its answer would change on a set of measure 0 relative to the current value. This is in keeping with the analysis of Section 3.3.3, where we curtailed some portion of the analysis as soon as we could tell that the answer didn't matter. Theorem 3.66 guarantees that there always *will* be a point at which things are irrelevant; the implementation is often able to terminate its reasoning before the conditions of the theorem are satisfied. Most of the time used by the planner is spent in reasoning of just this sort, deciding whether a particular line of reasoning might impact the plan being generated.

3.3.5 Summary

Our overall aim here has been to suggest that planners should manipulate not specialized plans that are guaranteed to achieve their goals, but more general plans that can only be expected to. We have presented a formalization of this idea of "expecting" a plan to achieve a goal in terms of the plan failing for a set of measure 0 in the set of its possible executions.

Building a planner around this idea introduces additional possibilities that existing planners lack; most important among these is that it is possible to combine approximate plans for each of two subgoals to obtain an approximate plan for their conjunction. Our main technical result in this area is Theorem 3.66, which confirms this observation. An examination of the tower-construction problem indicates that such a planner will have advantages over a conventional one in that it will debug plans constructed using independence assumptions as opposed to catering to all possible plan interactions at the outset.

We have implemented our ideas by exploiting the fact that plan sets can be viewed as elements of a bilattice. Some existing planners such as O-PLAN appear to make informal use of the ideas that we have discussed, but we know of no planner that explicitly conforms to the notions we have presented.

4 Heuristic results: Reason maintenance and dynamic backtracking

In this section of the report, we present additional technical results that were developed in the course of our research. Although the method described has not been incorporated into our planning system, it is generating substantial excitement in the AI community at large, and its inclusion in a summary report of the project seems warranted.

Dynamic backtracking is intended to address the problem that in spite of its intuitive appeal, the idea of maintaining and manipulating justification information has had surprisingly little impact on practical problem solving. Although there has been substantial theoretical work describing techniques for dealing with justifications, there are significant heuristic difficulties. Previous methods that maintained justification information stored a new justification with every backtrack [100, and subsequent work]. This means that the amount of memory required has been linear in the runtime, obviously inappropriate in practice. Dynamic backtracking achieves substantial computational gains while using far less memory.

To understand the method, suppose that we are trying to solve some constraint-satisfaction problem, or CSP. In the interests of definiteness, imagine that the CSP involves coloring a map of the United States subject to the restriction that adjacent states be colored differently.

Suppose we begin by coloring the states along the Mississippi, thereby splitting the remaining problem in two. We now begin to color the states in the western half of the country, coloring perhaps half a dozen of them before deciding that we are likely to be able to color the rest. Suppose also that the last state colored was Arizona.

At this point, we change our focus to the eastern half of the country. After all, if we can't color the eastern half because of our coloring choices for the states along the Mississippi, there is no point in wasting time completing the coloring of the western states.

We successfully color the eastern states and then return to the west. Unfortunately, we color New Mexico and Utah and then get stuck, unable to color (say) Nevada. What's more, backtracking doesn't help, at least in the sense that changing the colors for New Mexico and Utah alone does not allow us to proceed farther. Depth-first search would now have us backtrack to the eastern states, trying a new color for (say) New York in the vain hope that this would solve our problems out West.

This is obviously pointless; the blockade along the Mississippi makes it impossible for New York to have any impact on our attempt to color Nevada or other western states. What's more, we are likely to examine every *possible* coloring of the eastern states before addressing the problem that is actually the source of our difficulties.

The solutions that have been proposed to this involve finding ways to backtrack directly to some state that might actually allow us to make progress, in this case Arizona or earlier. Dependency-directed backtracking [100] involves a direct backtrack to the source of the difficulty; backjumping [29] avoids the computational overhead of this technique by using syntactic methods to estimate the point to which backtrack is necessary.

In both cases, however, note that although we backtrack to the source of the problem, we backtrack *over* our successful solution to half of the original problem, discarding our solution to the problem of coloring the states in the East. And once again, the problem is worse than this – after we recolor Arizona, we are in danger of solving the East yet again before realizing that our new choice for Arizona needs to be changed after all. We won't examine every possible coloring of the eastern states, but we are in danger of rediscovering our successful coloring an exponential number of times.

This hardly seems sensible; a human problem solver working on this problem would simply ignore the East if possible, returning directly to Arizona and proceeding. Only if the states along the Mississippi needed new colors would the East be reconsidered – and even then only if no new coloring could be found for the Mississippi that was consistent with the eastern solution.

In this section of the report we formalize this technique, presenting a modification to conventional search techniques that is capable of backtracking not only to the most recently expanded node, but also directly to a node elsewhere in the search tree. Because of the dynamic way in which the search is structured, we refer to this technique as *dynamic backtracking.*

A more specific outline is as follows: We begin by introducing a variety of notational conventions that allow us to cast both existing work and our new ideas in a uniform computational setting. Section 4.2 discusses backjumping, an intermediate between simple chronological backtracking and our ideas, which are themselves presented in Section 4.3. An example of the dynamic backtracking algorithm in use appears in Section 4.4 and an experimental analysis of the technique in Section 4.5. A summary of our results and suggestions for future work are in Section 4.6.

4.1 Preliminaries

Definition 4.1 *By a* constraint satisfaction problem (I, V, κ) *we will mean a set I of variables; for each $i \in I$, there is a set V_i of possible values for the variable i. κ is a set of constraints, each a pair (J, P) where $J = (j_1, \ldots, j_k)$ is an ordered subset of I and P is a subset of $V_{j_1} \times \cdots \times V_{j_k}$.*

A solution *to the* CSP *is a set v_i of values for each of the variables in I such that $v_i \in V_i$ for each i and for every constraint (J, P) of the above form in κ, $(v_{j_1}, \ldots, v_{j_k}) \in P$.*

In the previous example, I is the set of states and V_i is the set of possible colors for the state i. For each constraint, the first part of the constraint is a pair of adjacent states and the second part is a set of allowable color combinations for these states.

Our basic plan in this section of the report is to present formal versions of the search algorithms described earlier, beginning with simple depth-first search and proceeding to backjumping and dynamic backtracking. As a start, we make the following definition of a partial solution to a CSP:

Definition 4.2 *Let (I, V, κ) be a* CSP*. By a* partial solution *to the* CSP *we mean an ordered subset $J \subseteq I$ and an assignment of a value to each variable in J.*

We will denote a partial solution by a tuple of ordered pairs, where each ordered pair (i, v) assigns the value v to the variable i. For a partial solution P, we will denote by $\overline{P}$ the set of variables assigned values by P.

Constraint-satisfaction problems are solved in practice by taking partial solutions and extending them by assigning values to new variables. In general, of course, not any value can be assigned to a variable because some are inconsistent with the constraints. We therefore make the following definition:

Definition 4.3 *Given a partial solution P to a* CSP*, an* eliminating explanation *for a variable i is a pair (v, S) where $v \in V_i$ and $S \subseteq \overline{P}$. The intended meaning is that i cannot take the value v because of the values already assigned by P to the variables in S. An* elimination mechanism ϵ *for a* CSP *is a function that accepts as arguments a partial solution P, and a variable $i \notin \overline{P}$. The function returns a (possibly empty) set $\epsilon(P, i)$ of eliminating explanations for i.*

For a set E of eliminating explanations, we will denote by $\widehat{E}$ the values that have been identified as eliminated, ignoring the reasons given. We therefore denote by $\widehat{\epsilon}(P, i)$ the set of values eliminated by elements of $\epsilon(P, i)$.

Note that the above definition is somewhat flexible with regard to the amount of work done by the elimination mechanism – all values that violate completed constraints might be eliminated, or some amount of lookahead might be done. We will, however, make the following assumptions about all elimination mechanisms:

1. They are *correct.* For a partial solution P, if the value $v_i \notin \widehat{\epsilon}(P, i)$, then every constraint (S, T) in κ with $S \subseteq \overline{P} \cup \{i\}$ is satisfied by the values in the partial solution and the value v_i for i. These are the constraints that are complete after the value v_i is assigned to i.

2. They are *complete.* Suppose that P is a partial solution to a CSP, and there is some solution that extends P while assigning the value v to i. If P' is an extension of P with $(v, E) \in \epsilon(P', i)$, then
$$E \cap (\overline{P'} - \overline{P}) \neq \emptyset \tag{110}$$
In other words, whenever P can be successfully extended after assigning v to i but P' cannot be, at least one element of $P' - P$ is identified as a possible reason for the problem.

3. They are *concise*. For a partial solution P, variable i and eliminated value v, there is at most a single element of the form $(v, E) \in \epsilon(P, i)$. Only one reason is given why the variable i cannot have the value v.

Lemma 4.4 *Let ϵ be a complete elimination mechanism for a* CSP*, let P be a partial solution to this* CSP *and let $i \notin \overline{P}$. Now if P can be successfully extended to a complete solution after assigning i the value v, then $v \notin \hat{\epsilon}(P, i)$.*[34]

We apologize for the swarm of definitions, but they allow us to give a clean description of depth-first search:

Algorithm 4.5 (Depth-first search) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ϵ:*

1. *Set $P = \emptyset$. P is a partial solution to the* CSP*. Set $E_i = \emptyset$ for each $i \in I$; E_i is the set of values that have been eliminated for the variable i.*

2. *If $\overline{P} = I$, so that P assigns a value to every element in I, it is a solution to the original problem. Return it. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = \hat{\epsilon}(P, i)$, the values that have been eliminated as possible choices for i.*

3. *Set $S = V_i - E_i$, the set of remaining possibilities for i. If S is nonempty, choose an element $v \in S$. Add (i, v) to P, thereby setting i's value to v, and return to step 2.*

4. *If S is empty, let (j, v_j) be the last entry in P; if there is no such entry, return failure. Remove (j, v_j) from P, add v_j to E_j, set $i = j$ and return to step 3.*

We have written the algorithm so that it returns a single answer to the CSP; the modification to accumulate all such answers is straightforward.

The problem with Algorithm 4.5 is that it looks very little like conventional depth-first search, since instead of recording the unexpanded children of any particular node, we are keeping track of the *failed siblings* of that node. But we have the following:

Lemma 4.6 *At any point in the execution of Algorithm 4.5, if the last element of the partial solution P assigns a value to the variable i, then the unexplored siblings of the current node are those that assign to i the values in $V_i - E_i$.*

Proposition 4.7 *Algorithm 4.5 is equivalent to depth-first search and therefore complete.*

As we have remarked, the basic difference between Algorithm 4.5 and a more conventional description of depth-first search is the inclusion of the elimination sets E_i. The conventional description expects nodes to include pointers back to their parents; the siblings of a given

[34] As usual, proofs appear elsewhere [40].

node are found by examining the children of that node's parent. Since we will be reorganizing the space as we search, this is impractical in our framework.

It might seem that a more natural solution to this difficulty would be to record not the values that have been *eliminated* for a variable i, but those that remain to be considered. The technical reason that we have not done this is that it is much easier to maintain elimination information as the search progresses. To understand this at an intuitive level, note that when the search backtracks, the conclusion that has implicitly been drawn is that a particular node fails to expand to a solution, as opposed to a conclusion about the currently unexplored portion of the search space. It should be little surprise that the most efficient way to manipulate this information is by recording it in approximately this form.

4.2 Backjumping

How are we to describe dependency-directed backtracking or backjumping in this setting? In these cases, we have a partial solution and have been forced to backtrack; these more sophisticated backtracking mechanisms use information about the *reason* for the failure to identify backtrack points that might allow the problem to be addressed. As a start, we need to modify Algorithm 4.5 to maintain the explanations for the eliminated values:

Algorithm 4.8 *Given as inputs a constraint-satisfaction problem and an elimination mechanism ϵ:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$. E_i is a set of eliminating explanations for i.*

2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = \epsilon(P, i)$.*

3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i, v) to P and return to step 2.*

4. *If S is empty, let (j, v_j) be the last entry in P; if there is no such entry, return failure. Remove (j, v_j) from P. We must have $\widehat{E}_i = V_i$, so that every value for i has been eliminated; let E be the set of all variables appearing in the explanations for each eliminated value. Add $(v_j, E - \{j\})$ to E_j, set $i = j$ and return to step 3.*

Lemma 4.9 *Let P be a partial solution obtained during the execution of Algorithm 4.8, and let $i \in \overline{P}$ be a variable assigned a value by P. Now if $P' \subseteq P$ can be successfully extended to a complete solution after assigning i the value v but $(v, E) \in E_i$, we must have*

$$E \cap (\overline{P} - \overline{P'}) \neq \emptyset$$

In other words, the assignment of a value to some variable in $\overline{P}-\overline{P'}$ is correctly identified as the source of the problem.

Note that in step 4 of the algorithm, we could have added $(v_j, E \cap \overline{P})$ instead of $(v_j, E - \{j\})$ to E_j; either way, the idea is to remove from E any variables that are no longer assigned values by P.

In backjumping, we now simply change our backtrack method; instead of removing a single entry from P and returning to the variable assigned a value prior to the problematic variable i, we return to a variable that has actually had an impact on i. In other words, we return to some variable in the set E.

Algorithm 4.10 (Backjumping) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ϵ:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$.*

2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = \epsilon(P, i)$.*

3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i, v) to P and return to step 2.*

4. *If S is empty, we must have $\widehat{E}_i = V_i$. Let E be the set of all variables appearing in the explanations for each eliminated value.*

5. *If $E = \emptyset$, return failure. Otherwise, let (j, v_j) be the last entry in P such that $j \in E$. Remove from P this entry and any entry following it. Add $(v_j, E \cap \overline{P})$ to E_j, set $i = j$ and return to step 3.*

In step 5, we add $(v_j, E \cap \overline{P})$ to E_j, removing from E any variables that are no longer assigned values by P.

Proposition 4.11 *Backjumping is complete and always expands fewer nodes than does depth-first search.*

Let us have a look at this in our map-coloring example. If we have a partial coloring P and are looking at a specific state i, suppose that we denote by C the set of colors that are obviously illegal for i because they conflict with a color already assigned to one of i's neighbors.

One possible elimination mechanism returns as $\epsilon(P, i)$ a list of $(c, \overline{P})$ for each color $c \in C$ that has been used to color a neighbor of i. This reproduces depth-first search, since we gradually try all possible colors but have no idea what went wrong when we need to backtrack since every colored state is included in $\overline{P}$. A far more sensible choice would take $\epsilon(P, i)$ to be a list of $(c, \{n\})$ where n is a neighbor that is already colored c. This would ensure that we backjump to a neighbor of i if no coloring for i can be found.

If this causes us to backjump to another state j, we will add i's neighbors to the eliminating explanation for j's original color, so that if we need to backtrack still further, we consider neighbors of either i *or* j. This is as it should be, since changing the color of one of i's other neighbors might allow us to solve the coloring problem by reverting to our original choice of color for the state j.

We also have:

Proposition 4.12 *The amount of space needed by backjumping is $o(i^2v)$, where $i = |I|$ is the number of variables in the problem and v is the number of values for that variable with the largest value set V_i.*

This result contrasts sharply with an approach to CSPs that relies on truth-maintenance techniques to maintain a list of nogoods [10]. As we have already remarked, the number of nogoods found can grow linearly with the time taken for the analysis, and this will typically be exponential in the size of the problem. Backjumping avoids this problem by resetting the set E_i of eliminating explanations in step 2 of Algorithm 4.10.

The description that we have given is quite similar to that developed by Bruynooghe [5]. The explanations there are somewhat coarser than ours, listing all of the variables that have been involved in *any* eliminating explanation for a particular variable in the CSP, but the idea is essentially the same. Bruynooghe's eliminating explanations can be stored in $o(i^2)$ space (instead of $o(i^2v)$), but the associated loss of information makes the technique less effective in practice. This earlier work is also a description of backjumping only, since intermediate information is erased as the search proceeds.

4.3 Dynamic backtracking

We finally turn to new results. The basic problem with Algorithm 4.10 is not that it backjumps to the wrong place, but that it needlessly erases a great deal of the work that has been done thus far. At the very least, we can retain the values selected for variables that are backjumped over, in some sense moving the backjump variable to the end of the partial solution in order to replace its value without modifying the values of the variables that followed it.

There is an additional modification that will probably be clearest if we return to the example of the introduction. Suppose that in this example, we color only *some* of the eastern states before returning to the western half of the country. We reorder the variables in order to backtrack to Arizona and eventually succeed in coloring the West without disturbing the colors used in the East.

Unfortunately, when we return East backtracking is required and we find ourselves needing to change the coloring on some of the eastern states with which we dealt earlier. The ideas that we have presented will allow us to avoid erasing our solution to the problems out West, but if the search through the eastern states is to be efficient, we will need to retain the information we have about the portion of the East's search space that has been eliminated. After all, if we have determined that New York cannot be colored yellow, our changes in the

West will not reverse this conclusion – the Mississippi really does isolate one section of the country from the other.

The machinery needed to capture this sort of reasoning is already in place. When we backjump over a variable k, we should retain not only the choice of value for k, but also k's elimination set. We do, however, need to remove from this elimination set any entry that involves the eventual backtrack variable j, since these entries are no longer valid – they depend on the assumption that j takes its old value, and this assumption is now false.

Algorithm 4.13 (Dynamic backtracking I) *Given as inputs a constraint-satisfaction problem and an elimination mechanism ϵ:*

1. *Set $P = E_i = \emptyset$ for each $i \in I$.*
2. *If $\overline{P} = I$, return P. Otherwise, select a variable $i \in I - \overline{P}$. Set $E_i = E_i \cup \epsilon(P,i)$.*
3. *Set $S = V_i - \widehat{E}_i$. If S is nonempty, choose an element $v \in S$. Add (i,v) to P and return to step 2.*
4. *If S is empty, we must have $\widehat{E}_i = V_i$; let E be the set of all variables appearing in the explanations for each eliminated value.*
5. *If $E = \emptyset$, return failure. Otherwise, let (j,v_j) be the last entry in P such that $j \in E$. Remove (j,v_j) from P and, for each variable k assigned a value after j, remove from E_k any eliminating explanation that involves j. Set*

$$E_j = E_j \cup \epsilon(P,j) \cup \{(v_j, E \cap \overline{P})\} \tag{111}$$

so that v_j is eliminated as a value for j because of the values taken by variables in $E \cap \overline{P}$. The inclusion of the term $\epsilon(P,j)$ incorporates new information from variables that have been assigned values since the original assignment of v_j to j. Now set $i = j$ and return to step 3.

Theorem 4.14 *Dynamic backtracking always terminates and is complete. It continues to satisfy Proposition 4.12 and can be expected to expand fewer nodes than backjumping provided that the goal nodes are distributed randomly in the search space.*

The essential difference between dynamic and dependency-directed backtracking is that the structure of our eliminating explanations means that we only save nogood information based on the current values of assigned variables; if a nogood depends on outdated information, we drop it. By doing this, we avoid the need to retain an exponential amount of nogood information. What makes this technique valuable is that (as stated in the theorem) termination is still guaranteed.

There is one trivial modification that we can make to Algorithm 4.13 that is quite useful in practice. After removing the current value for the backtrack variable j, Algorithm 4.13 immediately replaces it with another. But there is no real reason to do this; we could instead pick a value for an entirely different variable:

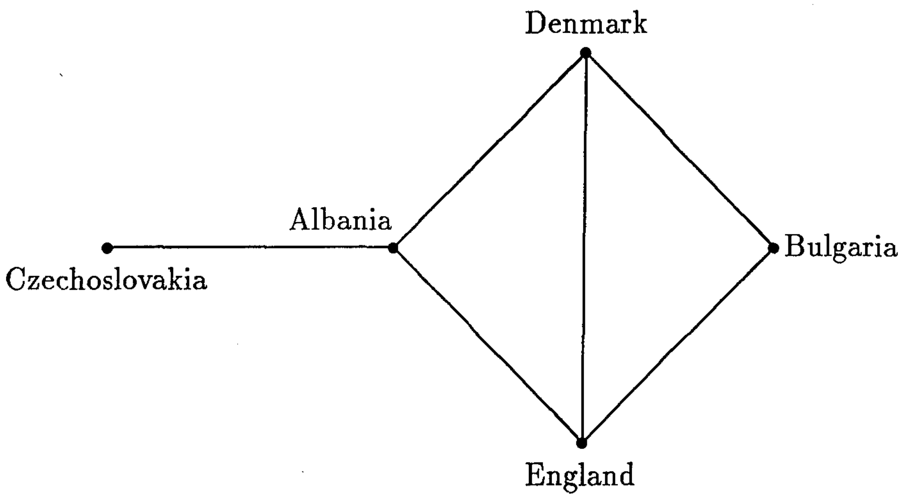


Figure 12: A small map-coloring problem

Algorithm 4.15 (Dynamic backtracking) *Given as inputs a constraint-satisfaction problem and an elimination mechanism* ϵ*:*

1. *Set* $P = E_i = \emptyset$ *for each* $i \in I$.
2. *If* $\overline{P} = I$*, return* P*. Otherwise, select a variable* $i \in I - \overline{P}$*. Set* $E_i = E_i \cup \epsilon(P, i)$.
3. *Set* $S = V_i - \widehat{E}_i$*. If* S *is nonempty, choose an element* $v \in S$*. Add* (i, v) *to* P *and return to step 2.*
4. *If* S *is empty, we must have* $\widehat{E}_i = V_i$*; let* E *be the set of all variables appearing in the explanations for each eliminated value.*
5. *If* $E = \emptyset$*, return failure. Otherwise, let* (j, v_j) *be the last entry in* P *that binds a variable appearing in* E*. Remove* (j, v_j) *from* P *and, for each variable* k *assigned a value after* j*, remove from* E_k *any eliminating explanation that involves* j*. Add* $(v_j, E \cap \overline{P})$ *to* E_j *and return to step 2.*

4.4 An example

In order to make Algorithm 4.15 a bit clearer, suppose that we consider a small map-coloring problem in detail. The map is shown in Figure 12 and consists of five countries: Albania, Bulgaria, Czechoslovakia, Denmark and England. We will assume (wrongly!) that the countries border each other as shown in the figure, where countries are denoted by nodes and border one another if and only if there is an arc connecting them.

In coloring the map, we can use the three colors red, yellow and blue. We will typically abbreviate the country names to single letters in the obvious way.

We begin our search with Albania, deciding (say) to color it red. When we now look at Bulgaria, no colors are eliminated because Albania and Bulgaria do not share a border; we decide to color Bulgaria yellow. (This is a mistake.)

We now go on to consider Czechoslovakia; since it borders Albania, the color red is eliminated. We decide to color Czechoslovakia blue and the situation is now this:

country	color	red	yellow	blue
Albania	red			
Bulgaria	yellow			
Czechoslovakia	blue	A		
Denmark				
England				

For each country, we indicate its current color and the eliminating explanations that mean it cannot be colored each of the three colors (when such explanations exist). We now look at Denmark.

Denmark cannot be colored red because of its border with Albania and cannot be colored yellow because of its border with Bulgaria; it must therefore be colored blue. But now England cannot be colored any color at all because of its borders with Albania, Bulgaria and Denmark, and we therefore need to backtrack to one of these three countries. At this point, the elimination lists are as follows:

country	color	red	yellow	blue
Albania	red			
Bulgaria	yellow			
Czechoslovakia	blue	A		
Denmark	blue	A	B	
England		A	B	D

We backtrack to Denmark because it is the most recent of the three possibilities, and begin by removing any eliminating explanation involving Denmark from the above table to get:

country	color	red	yellow	blue
Albania	red			
Bulgaria	yellow			
Czechoslovakia	blue	A		
Denmark		A	B	
England		A	B	

Next, we add to Denmark's elimination list the pair

$$(\text{blue}, \{A, B\})$$

This indicates correctly that because of the current colors for Albania and Bulgaria, Denmark cannot be colored blue (because of the subsequent dead end at England). Since every color is now eliminated, we must backtrack to a country in the set $\{A, B\}$. Changing Czechoslovakia's color won't help and we must deal with Bulgaria instead. The elimination lists are now:

country	color	red	yellow	blue
Albania	red			
Bulgaria				
Czechoslovakia	blue	A		
Denmark		A	B	A,B
England		A	B	

We remove the eliminating explanations involving Bulgaria and also add to Bulgaria's elimination list the pair

$$(\text{yellow}, A)$$

indicating correctly that Bulgaria cannot be colored yellow because of the current choice of color for Albania (red).

The situation is now:

country	color	red	yellow	blue
Albania	red			
Czechoslovakia	blue	A		
Bulgaria			A	
Denmark		A		
England		A		

We have moved Bulgaria past Czechoslovakia to reflect the search reordering in the algorithm. We can now complete the problem by coloring Bulgaria red, Denmark either yellow or blue, and England the color not used for Denmark.

This example is almost trivially simple, of course; the thing to note is that when we changed the color for Bulgaria, we retained both the blue color for Czechoslovakia and the information indicating that none of Czechoslovakia, Denmark and England could be red. In more complex examples, this information may be very hard-won and retaining it may save us a great deal of subsequent search effort.

Another feature of this specific example (and of the example of the introduction as well) is that the computational benefits of dynamic backtracking are a consequence of the automatic realization that the problem splits into disjoint subproblems. Other authors have also discussed the idea of applying divide-and-conquer techniques to CSPs [86, 107], but their methods suffer from the disadvantage that they constrain the order in which unassigned variables are assigned values, perhaps at odds with the common heuristic of assigning values first to those variables that are most tightly constrained. Dynamic backtracking can also be expected to be of use in situations where the problem in question does *not* split into two or more disjoint subproblems.[35]

[35] Research conducted outside the scope of this project has since confirmed this [48].

Frame	Dynamic backtracking	Backjumping	Frame	Dynamic backtracking	Backjumping
1	100	100	11	100	98
2	100	100	12	100	100
3	100	100	13	100	100
4	100	100	14	100	100
5	100	100	15	99	14
6	100	100	16	100	26
7	100	100	17	100	30
8	100	100	18	61	0
9	100	100	19	10	0
10	100	100			

Figure 13: Number of problems solved successfully

4.5 Experimentation

Dynamic backtracking has been incorporated into a successful crossword-puzzle generation program [44], and leads to significant performance improvements in that restricted domain. More specifically, the method was tested on the problem of generating 19 puzzles of sizes ranging from 2 × 2 to 13 × 13; each puzzle was attempted 100 times using both dynamic backtracking and simple backjumping. The dictionary was shuffled between solution attempts and a maximum of 1000 backtracks were permitted before the program was deemed to have failed.

In both cases, the algorithms were extended to include iterative broadening [46], the cheapest-first heuristic and forward checking. Cheapest-first has also been called "most constrained first" and selects for instantiation that variable with the fewest number of remaining possibilities (i.e., that variable for which it is cheapest to enumerate the possible values [95]). Forward checking prunes the set of possibilities for crossing words whenever a new word is entered and constitutes our experimental choice of elimination mechanism: at any point, words for which there is no legal crossing word are eliminated. This ensures that no word will be entered into the crossword if the word has *no* potential crossing words at some point. The cheapest-first heuristic would identify the problem at the next step in the search, but forward checking reduces the number of backtracks substantially. The "least-constraining" heuristic [44] was *not* used; this heuristic suggests that each word slot be filled with the word that minimally constrains the subsequent search. The heuristic was not used because it would invalidate the technique of shuffling the dictionary between solution attempts in order to gather useful statistics.

The table in Figure 13 indicates the number of successful solution attempts (out of 100) for each of the two methods on each of the 19 crossword frames. Dynamic backtracking is more successful in six cases and less successful in none.

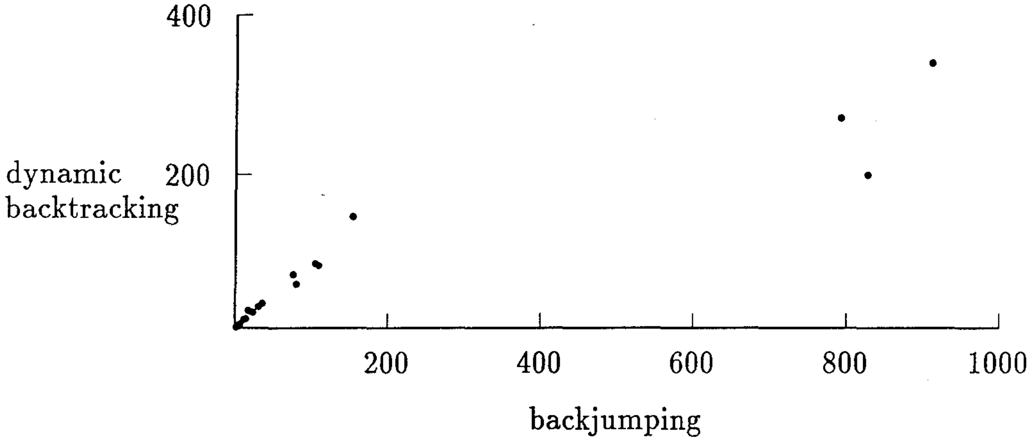


Figure 14: Number of backtracks needed

With regard to the number of nodes expanded by the two methods, consider the data presented in Figure 14, where we graph the average number of backtracks needed by the two methods.[36] Although initially comparable, dynamic backtracking provides increasing computational savings as the problems become more difficult. A somewhat broader set of experiments described elsewhere [55] leads to similar conclusions.

There are some examples where dynamic backtracking leads to performance degradation, however [55]; a typical case appears in Figure 15.[37] In this figure, we first color A, then B, then the countries in region 1, and then get stuck in region 2.

We now presumably backtrack directly to B, leaving the coloring of region 1 alone. But this may well be a mistake – the colors in region 1 will restrict our choices for B, perhaps making the subproblem consisting of A, B and region 2 more difficult than it might be. If region 1 were easy to color, we would have been better off erasing it even though we didn't need to. This topic has been explored further by other researchers [3].

4.6 Summary

4.6.1 Why it works

There are two separate ideas that we have exploited in the development of Algorithm 4.15 and the others leading up to it. The first, and easily the most important, exploits the similarity between our use of eliminating explanations and the use of nogoods in the ATMS community; the principal difference is that we attach the explanations to the variables they impact and drop them when they cease to be relevant. (They might become relevant again later, of course.) This avoids the prohibitive space requirements of systems that permanently cache the results of their nogood calculations, an observation that may be extensible beyond

[36] Only 17 points are shown because no point is plotted where backjumping was unable to solve the problem.

[37] The worst performance degradation observed was a factor of approximately 4.

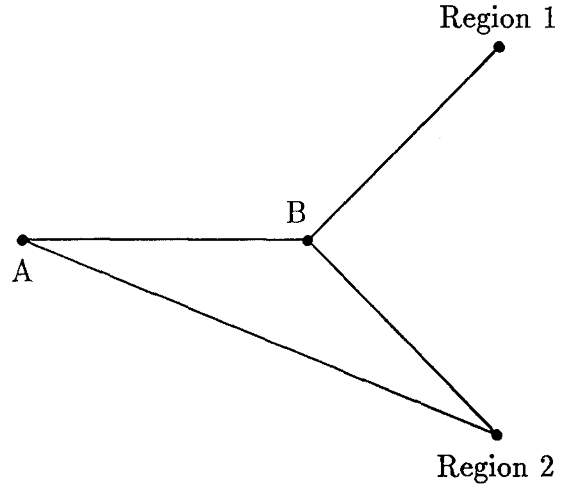


Figure 15: A difficult problem for dynamic backtracking

the domain of CSPs specifically. There are other ways to view this – Gashnig's notion of *backmarking* [29] records similar information about the reason that particular portions of a search space are known not to contain solutions.

Our second idea involves the fact that it is possible to modify variable order on the fly in a way that allows us to retain the results of earlier work when backtracking to a variable that was assigned a value early in the search.

This reordering should not be confused with the work of authors who have suggested a dynamic choice among the variables that *remain* to be assigned values [14, 44, 73, 108]; we are instead reordering the variables that have *been* assigned values in the search thus far.

Another way to look at this idea is that we have found a way to "erase" the value given to a variable directly as opposed to backtracking to it. This idea has also been explored by Minton *et.al* [69] and by Selman *et.al* [90]; these authors also directly replace values assigned to variables in satisfiability problems. Unfortunately, the heuristic repair method used is incomplete because no dependency information is retained from one state of the problem solver to the next.

There is a third way to view this as well. The space that we are examining is really a graph, as opposed to a tree; we reach the same point by coloring Albania blue and then Bulgaria red as if we color them in the opposite order. When we decide to backjump from a particular node in the search space, we know that we need to back up until some particular property of that node ceases to hold – and the key idea is that by backtracking along a path *other than* the one by which the node was generated, we may be able to backtrack only slightly when we would otherwise need to retreat a great deal. This observation is interesting because it may well apply to problems other than CSPs. Unfortunately, it is not clear how to guarantee completeness for a search that discovers a node using one path and backtracks using another.

4.6.2 Future work

There are a variety of ways in which the techniques we have presented can be extended; in this section, we sketch a few of the more obvious ones.

Backtracking to older culprits One extension to our work involves lifting the restriction in Algorithm 4.15 that the variable erased always be the most recently assigned member of the set E.

In general, we cannot do this while retaining the completeness of the search. Consider the following example:

Imagine that our CSP involves three variables, x, y and z, that can each take the value 0 or 1. Further, suppose that this CSP has no solutions, in that after we pick any two values for x and for y, we realize that there is no suitable choice for z.

We begin by taking $x = y = 0$; when we realize the need to backtrack, we introduce the nogood

$$x = 0 \supset y \neq 0 \tag{112}$$

and replace the value for y with $y = 1$.

This fails, too, but now suppose that we were to decide to backtrack to x, introducing the new nogood

$$y = 1 \supset x \neq 0 \tag{113}$$

We change x's value to 1 and erase (112).

This also fails. We decide that y is the problem and change its value to 0, introducing the nogood

$$x = 1 \supset y \neq 1$$

but erasing (113). And when *this* fails, we are in danger of returning to $x = y = 0$, which we eliminated at the beginning of the example. This loop may cause a modified version of the dynamic backtracking algorithm to fail to terminate. When we drop (112) in favor of (113), we are no longer in a position to recover (112).

We can deal with this by placing conditions on the variables to which we choose to backtrack [48]. This work provides substantial freedom in the choice of backtrack point.

This freedom of backtrack raises an important question that has not yet been addressed in the literature: When backtracking to avoid a difficulty of some sort, to where should one backtrack?

Previous work has been constrained to backtrack no further than the most recent choice that might impact the problem in question; any other decision would be both incomplete and inefficient. Although an extension of Algorithm 4.15 need not operate under this restriction, we have given no indication of how the backtrack point should be selected.

There are several easily identified factors that can be expected to bear on this choice. The first is that there remains a reason to expect backtracking to chronologically recent choices to be the most effective – these choices can be expected to have contributed to the fewest eliminating explanations, and there is obvious advantage to retaining as many eliminating explanations as possible from one point in the search to the next. It is possible, however, to simply identify that backtrack point that affects the fewest number of eliminating explanations and to use that.

Alternatively, it might be important to backtrack to the choice point for which there will be as many new choices as possible; as an extreme example, if there is a variable i for which every value other than its current one has already been eliminated for other reasons, backtracking to i is guaranteed to generate another backtrack immediately and should probably be avoided if possible.

Finally, there is some measure of the "directness" with which a variable bears on a problem. If we are unable to find a value for a particular variable i, it is probably sensible to backtrack to a second variable that shares a constraint with i itself, as opposed to some variable that affects i only indirectly.

Although we do not know how these competing considerations are to be weighed, the framework we have developed is interesting because it allows us to work on this question. In more basic terms, we can now "debug" partial solutions to CSPs directly, moving laterally through the search space in an attempt to remain as close to a solution as possible. This sort of lateral movement seems central to human solution of difficult search problems, and it is encouraging to begin to understand it in a formal way.

Dependency pruning It is often the case that when one value for a variable is eliminated while solving a CSP, others are eliminated as well. As an example, in solving a scheduling problem a particular choice of time (say $t = 16$) may be eliminated for a task A because there then isn't enough time between A and a subsequent task B; in this case, all later times can obviously be eliminated for A as well.

Formalizing this can be subtle; after all, a later time for A isn't *uniformly* worse than an earlier time because there may be other tasks that need to precede A and making A later makes that part of the schedule easier. It's the problem with B alone that forces A to be earlier; once again, the analysis depends on the ability to maintain dependency information as the search proceeds.

We can formalize this as follows. Given a CSP (I, V, κ), suppose that the value v has been assigned to some $i \in I$. Now we can construct a new CSP (I', V', κ') involving the remaining variables $I' = I - \{i\}$, where the new set V' need not mention the possible values V_i for i, and where κ' is generated from κ by modifying the constraints to indicate that i has been assigned the value v. We also make the following definition:

Definition 4.16 *Given a* CSP*, suppose that i is a variable that has two possible values u and v. We will say that v is* stricter *than u if every constraint in the* CSP *induced by assigning u to i is also a constraint in the* CSP *induced by assigning i the value v.*

The point, of course, is that if v is stricter than u is, there is no point to trying a solution involving v once u has been eliminated. After all, finding such a solution would involve satisfying all of the constraints in the v restriction, these are a superset of those in the u restriction, and we were unable to satisfy the constraints in the u restriction originally.

The example with which we began this section now generalizes to the following:

Proposition 4.17 *Suppose that a* CSP *involves a set S of variables, and that we have a partial solution that assigns values to the variables in some subset $P \subseteq S$. Suppose further that if we extend this partial solution by assigning the value u to a variable $i \notin P$, there is no further extension to a solution of the entire* CSP. *Now consider the* CSP *involving the variables in $S - P$ that is induced by the choices of values for variables in P. If v is stricter than u as a choice of value for i in this problem, the original* CSP *has no solution that both assigns v to i and extends the given partial solution on P.*

This proposition isn't quite enough; in the earlier example, the choice of $t = 17$ for A will not be stricter than $t = 16$ if there is any task that needs to be scheduled before A is. We need to record the fact that B (which is no longer assigned a value) is the source of the difficulty. To do this, we need to augment the dependency information with which we are working.

More precisely, when we say that a set of variables $\{x_i\}$ eliminates a value v for a variable x, we mean that our search to date has allowed us to conclude that

$$(v_1 = x_1) \wedge \cdots \wedge (v_k = x_k) \supset v \neq x$$

where the v_i are the current choices for the x_i. We can obviously rewrite this as

$$(v_1 = x_1) \wedge \cdots \wedge (v_k = x_k) \wedge (v = x) \supset F \tag{114}$$

where F indicates that the CSP in question has no solution.

Let's be more specific still, indicating in (114) exactly *which* CSP has no solution:

$$(v_1 = x_1) \wedge \cdots \wedge (v_k = x_k) \wedge (v = x) \supset F(I) \tag{115}$$

where I is the set of variables in the complete CSP.

Now we can address the example with which we began this section; the CSP that is known to fail in an expression such as (115) is not the entire problem, but only a subset of it. In the example, we are considering, the subproblem involves only the two tasks A and B. In general, we can augment our nogoods to include information about the subproblems on which they fail, and then measure strictness with respect to these restricted subproblems only. In our example, this will indeed allow us to eliminate $t = 17$ from consideration as a possible time for A.

The additional information stored with the nogoods doubles their size (we have to store a second subset of the variables in the CSP), and the variable sets involved can be manipulated easily as the search proceeds. The cost involved in employing this technique is therefore that of the strictness computation. This may be substantial given the data structures currently used to represent CSPs (which typically support the need to check if a constraint has been violated but little more), but it seems likely that compile-time modifications to these data structures can be used to make the strictness question easier to answer. In scheduling problems, preliminary experimental work shows that the idea is an important one; here, too, there is much to be done.

The basic lesson of dynamic backtracking is that by retaining only those nogoods that are still relevant given the partial solution with which we are working, the storage difficulties encountered by full dependency-directed methods can be alleviated. This is what makes all of the ideas we have proposed possible – erasing values, selecting alternate backtrack points, and dependency pruning. There are surely many other effective uses for a practical dependency maintenance system as well.

5 Conclusion

5.1 Scientific contributions

This project led to three major technical advances during the contract period. Approximate planning and dynamic backtracking promise to substantially change the way various AI subcommunities solve problems. The contributions to anytime declarative reasoning also appear likely to lead to significant technical and implementation progress, but are less revolutionary.

5.1.1 Anytime reasoning

Problem: Declarative problem-solving methods cannot be interrupted; they run for an extended amount of time (since they are typically solving intractable problems) and then return an answer. This is inappropriate in real-time applications such as those that are the focus of the transportation planning initiative.

Previous solution attempts: Many. Typical are Dean and Boddy's [11] proposal of a framework but no real mechanism and Elgot-Drapkin and Perlis's [19] description of a brittle mechanism that depends critically on the choice of language used in any particular application and is likely to be unstable in realistic systems.

Key idea: Use modal operators already present in the declarative database as markers for points at which inference can be suspended and a meaningful partial answer returned. Computation can be resumed if better answers are needed and more time is available for analysis. Because modal operators have well-defined meanings independent of language choice, this overall approach is both more robust and more stable than previous work.

Impact: In addition to its intrinsic merits, this work is enabling technology. It led directly, for example, to the development of approximate planning. It should also lead to a better general methodology for fielding declarative systems in real-time environments.

5.1.2 Approximate planning

Problem: Existing generative planning tools are unable to debug complex plans in the face of new information or execution difficulties. They cannot reason in real time, do not support such crucial objectives as economy of force, serendipity, component reuse and parallelization, and cannot be used in conjunction with human input (so-called *mixed-initiative* planning).

Previous solution attempts: The planning community recognizes that the above difficulties can largely be addressed by a system capable of solving subgoals separately and then merging the results. Unfortunately, no principled planning system functions in this fashion. Tate's O-PLAN system [9] separates conjuncts but there is no justification for the mechanism used. Kambhampati [57] discusses the need for plan debugging but is unable to address separate conjuncts individually. Plan merging is discussed by Foulser *et.al* [28] but no method is presented for incorporating the technique directly into a planner. The best known formal methods make no use of plan merging whatsoever [54, 65].

Key idea: It is possible to plan for conjuncts separately if (and only if) one allows for the possibility that the resulting subplans may interfere when merged. The approximate planning work formalizes this idea and develops precise analyses that bound the importance of the inaccuracies the merge may introduce. It also provides a mechanism whereby the inaccuracies can be reduced if additional computational resources are available.

Impact: Approximate planning should substantially change the AI community's approach to planning generally, and contribute to the use of well-founded generative planners in practical applications.

5.1.3 Dynamic backtracking

Problem: Recording the reason that certain decisions were made can improve the efficiency of tools that solve constraint-satisfaction problems such as scheduling. Unfortunately, keeping such justifications has required an amount of memory linear in the run time, and therefore exponential in the problem size. This has made justification-based methods useless in practice.

Previous solution attempts: None. It was widely believed that no polynomial-space justification-based algorithm could exist.

Key idea: By retaining only those justifications that constrain the current partial solution, an algorithm can be developed that is guaranteed to find a solution if one exists while using only a polynomial amount of memory.

Impact: Several other authors are now pursuing this line of work. Honeywell, for example, has used the ideas to improve the speed of one of their scheduling tools by an order of magnitude.

5.2 Transportation planning contributions

In this final section, we focus on the impact that our work can be expected to have on the transportation initiative specifically. Because our focus here is on reasonably short-term payoffs, we will concentrate on dynamic backtracking (Section 5.2.1) and approximate planning (Section 5.2.2). The impact of anytime declarativism is discussed in Section 5.2.2 as well.

5.2.1 Dynamic backtracking

Any effective scheduling system for military operations will need to examine the space of possible schedules locally, beginning with a specific candidate schedule and then examining possibilities that are "nearby" in some suitable sense. This may be because the search space itself is so large that it is impractical to examine it in its entirety, or may be because execution failures or other contingencies force the plan to be modified in real time after construction.

The participants in the transportation planning initiative recognize the problem, at least in a limited way. The two most successful scheduling algorithms developed in the initiative [98, 99] do no search at all, simply returning the best schedule that can be constructed without backtracking. It is well known, however, that local search can improve the results of single-probe methods such as these [69, and others].

The key to making local search methods practical on large problems is the effective use of justification information. Justification information is essential if the search is to be systematic, and also appears crucial if fast local search techniques such as GSAT [88, 89, 90] are to be applied to realistic domains.

Unfortunately, maintaining justification information has in the past not been feasible for large systems because the memory required is linear in the run time. The conventional wisdom in the search community has been that no subexponential set of justifications could be used to usefully control the search.

This conventional wisdom is simply wrong; dynamic backtracking is the proof. Because it is the first principled mechanism that achieves significant computational advantage while keeping only polynomially many justifications, dynamic backtracking promises to significantly enhance our abilities to construct solutions to large-scale scheduling (or other constraint-satisfaction) problems, and to then modify these solutions locally as execution difficulties warrant.

5.2.2 Approximate planning

ARPA and Rome Labs recently commissioned a group of scientists ("Swat Team B") to identify specific functionality that would be required of a visionary transportation planning system. The report of Swat Team B includes a specific air campaign planning scenario and a planning system ADEPT capable of providing decision support as follows:

1. ADEPT serves as a database, providing information about recent intelligent reports regarding potential targets of a hypothetical military strike.
2. ADEPT uses this information to construct high level plans for achieving operational goals.
3. ADEPT evaluates the likely effectiveness of these plans given specific assumptions about the targets' military condition.
4. ADEPT accepts suggestions regarding the selection of high level goals and constructs plans for achieving them. It provides an analysis of the differences between competing operational approaches, including a description of the assumptions underlying this analysis.
5. ADEPT modifies the operational plan to incorporate new intelligence information. This includes recognizing when the original operational plans are insufficient in the face of this new data.
6. Finally, ADEPT elaborates the operational plan to include considerations due to aircraft load, range, cruise speed, etc.

The planning needs of a system such as ADEPT imply that:

- ADEPT must be able to plan for subgoals separately. In order to satisfy the mixed-initiative requirement that ADEPT be able to accept advice and thereby focus its attention on specific subgoals selected by the user, ADEPT must be able to treat these subgoals separately during the planning process.
- ADEPT must record plan justifications and make them available to the user as part of its general explanation facility.
- If ADEPT is to replan in real time, the underlying planning system must itself exhibit real-time functionality.

A planning system based on the ideas in this report can meet these needs. We described in Section 3 a system that plans for subgoals separately, and can do so in real time. Conclusions regarding plan sets that do and do not achieve various subgoals serve as the justification information that the system is required to make available to its users. Indeed, given the philosophical arguments with which we began Section 3.3, it is difficult to imagine any other architecture for a military planning system that would serve as well.

References

[1] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, 1987.

[2] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–142. Morgan Kaufmann, 1987.

[3] A. B. Baker. The hazards of fancy backtracking. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[4] A. B. Baker and M. L. Ginsberg. Temporal projection and explanation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 906–911, 1989.

[5] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[6] D. Chan. Constructive negation based on the completed database. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 111–125, 1988.

[7] D. Chan. An extension of constructive negation and its application in coroutining. In *Logic Programming: Proceedings of the Sixth International Conference and Symposium*, 1989.

[8] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[9] K. Currie and A. Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[10] J. de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986.

[11] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[12] T. Dean and K. Kanazawa. Probabilistic temporal reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 524–528, 1988.

[13] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[14] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 271–277, 1989.

[15] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[16] M. Drummond. Situated control rules. Technical report, NASA Ames Research Center, Moffett Field, CA, 1988.

[17] M. Drummond and J. Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 138–144, 1990.

[18] J. J. Elgot-Drapkin. *Step-Logic: Reasoning Situated in Time.* PhD thesis, University of Maryland, College Park, MD, 1990.

[19] J. J. Elgot-Drapkin and D. Perlis. Reasoning in time I: Basic concepts. *J. Expt. Theor. Artif. Intell.*, 2:75–98, 1990.

[20] C. Elkan. Incremental, approximate planning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 145–150, 1990.

[21] L. Erman and V. Lesser. A multi-level organization for problem-solving using many diverse, cooperating sources of knowledge. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 483–490, Tbilisi,USSR, Aug. 1975.

[22] D. Etherington and J. Crawford. Toward efficient default reasoning. Unpublished manuscript, 1992.

[23] O. Etzioni. Why PRODIGY/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 916–922, 1990.

[24] R. Feldman and P. Morris. Admissible criteria for loop control in planning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 151–157, 1990.

[25] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[26] J. J. Finger. *Exploiting Constraints in Design Synthesis.* PhD thesis, Stanford University, Stanford, CA, 1987.

[27] M. C. Fitting. Logic programming on a topological bilattice. *Fundamenta Informatica*, 11:209–218, 1988.

[28] D. E. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57:143–181, 1992.

[29] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[30] M. Gelfond. On stratified autoepistemic theories. Technical report, University of Texas at El Paso, 1986.

[31] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[32] M. L. Ginsberg. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, San Mateo, CA, 1987.

[33] M. L. Ginsberg. Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[34] M. L. Ginsberg. A circumscriptive theorem prover. *Artificial Intelligence*, 39:209–230, 1989.

[35] M. L. Ginsberg. Bilattices and modal operators. *Journal of Logic and Computation*, 1:41–69, 1990.

[36] M. L. Ginsberg. The computational value of nonmonotonic reasoning. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Boston, MA, 1991.

[37] M. L. Ginsberg. The MVL theorem proving system. *SIGART Bulletin*, 2(3):57–60, 1991.

[38] M. L. Ginsberg. Negative subgoals with free variables. *Journal of Logic Programming*, 11:271–293, 1991.

[39] M. L. Ginsberg. What is the modal truth criterion? Technical report, Stanford University, 1991.

[40] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[41] M. L. Ginsberg. User's guide to the MVL system. Technical report, University of Oregon, 1993.

[42] M. L. Ginsberg. Modality and interrupts. *J. Approx. Reasoning*, 1994. Submitted.

[43] M. L. Ginsberg. Approximate planning. *Artificial Intelligence*, 1995.

[44] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 210–215, 1990.

[45] M. L. Ginsberg and D. F. Geddis. Is there any need for domain-dependent control information? In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[46] M. L. Ginsberg and W. D. Harvey. Iterative broadening. *Artificial Intelligence*, 55:367–383, 1992.

[47] M. L. Ginsberg and H. W. Holbrook. What defaults can do that hierarchies can't. In *Proceedings 1992 Nonmonotonic Reasoning Workshop*, Plymouth, VT, 1992.

[48] M. L. Ginsberg and D. A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Bonn, Germany, 1994.

[49] M. L. Ginsberg and D. E. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[50] C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 183–205. American Elsevier, New York, 1969.

[51] A. Haas. The case for domain-specific frame axioms. In F. M. Brown, editor, *The Frame Problem in Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1987.

[52] K. J. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228, 1990.

[53] S. Hanks and D. McDermott. Nonmonotonic logics and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[54] S. Hanks and D. S. Weld. Systematic adaptation for case-based planning. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, pages 96–105. Morgan Kaufmann, 1992.

[55] A. K. Jonsson and M. L. Ginsberg. Experimenting with new systematic and nonsystematic search techniques. In *Proceedings of the AAAI Spring Symposium on AI and NP-Hard Problems*, Stanford, California, 1993.

[56] L. P. Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 60–65, 1988.

[57] S. Kambhampati. Design tradeoffs in partial order (plan space) planning. In *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference*, pages 92–87. AAAI Press, 1994.

[58] C. A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, 1990.

[59] K. Konolige. On the relation between default theories and autoepistemic logic. *Artificial Intelligence*, 35:343–382, 1988.

[60] K. Konolige. On the relation between default theories and autoepistemic logic (erratum). *Artificial Intelligence*, 41:115, 1990.

[61] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[62] S. A. Kripke. Semantical considerations on modal logic. In L. Linsky, editor, *Reference and Modality*, pages 63–72. Oxford University Press, London, 1971.

[63] V. Lifschitz. On the semantics of STRIPS. In *Proceedings of the 1986 Workshop on Planning and Reasoning about Action*, Timberline, Oregon, 1986.

[64] B. L. Lipman. How to decide how to decide how to ...: Modeling limited rationality. *Econometrica*, 1991.

[65] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639, 1991.

[66] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[67] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[68] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[69] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 17–24, 1990.

[70] R. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[71] K. L. Myers and D. E. Smith. On the persistence of derived beliefs. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988.

[72] N. J. Nilsson. Action networks. Technical report, Stanford University, 1988.

[73] C. B. P. Purdom and E. Robertson. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–114, 1981.

[74] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–113, Boston, MA, 1992.

[75] D. Perlis. Non-monotonicity and real-time reasoning. In *Proceedings 1984 Nonmonotonic Reasoning Workshop*, pages 363–372, New Paltz, NY, 1984. American Association for Artificial Intelligence.

[76] D. Poole, R. Aleliunas, and R. Goebel. THEORIST: A logical reasoning system for defaults and diagnosis. Technical report, University of Waterloo, 1985.

[77] T. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1987.

[78] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[79] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, Boston, 1991.

[80] R. Reiter and G. Criscuolo. On interacting defaults. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 270–276, 1981.

[81] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 183–188, 1987.

[82] S. J. Rosenschein and L. P. Kaelbling. The synthesis of machines with provable epistemic properties. In J. Y. Halpern, editor, *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 83–98, Los Altos, CA, 1986. Morgan Kaufmann.

[83] E. D. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, New York, 1977.

[84] M. J. Schoppers. Universal plans for reactive robots in unpredictable domains. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[85] L. K. Schubert. Monotonic solution of the frame problem in the situation calculus. In H. E. Kyburg, Jr., R. P. Loui, and G. N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23-67. Kluwer, Boston, 1990.

[86] R. Seidel. A new method for solving constraint satisfaction problems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 338-342, 1981.

[87] B. Selman and H. Kautz. Knowledge compilation using Horn approximations. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 904-909, 1991.

[88] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 290-295, 1993.

[89] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proceedings 1993 DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*, 1993.

[90] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440-446, 1992.

[91] B. Silver. *Meta-Level Inference.* Elsevier, New York, 1986.

[92] B. Skyrms. *Choice and Chance: An Introduction to Inductive Logic.* Dickerson, 1966.

[93] D. E. Smith. *Controlling Inference.* PhD thesis, Stanford University, Aug. 1985.

[94] D. E. Smith. Controlling backward inference. Knowledge Systems Laboratory Report 86-68, Stanford University, June 1986.

[95] D. E. Smith and M. R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26(2):171-215, 1985.

[96] D. E. Smith, M. R. Genesereth, and M. L. Ginsberg. Controlling recursive inference. *Artificial Intelligence*, 30:343-389, 1986.

[97] D. E. Smith and M. Peot. A critical look at Knoblock's hierarchy mechanism. Unpublished manuscript, 1992.

[98] D. R. Smith. Transformational approach to scheduling. Technical Report KES.U.92.2, Kestrel Institute, 1992.

[99] S. F. Smith and C.-C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 139-144, 1993.

[100] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[101] G. J. Sussman. *A Computational Model of Skill Acquisition.* American Elsevier, New York, 1975.

[102] A. Tate. Project planning using a hierarchic non-linear planner. Technical Report 25, Depart of Artificial Intelligence, University of Edinburgh, 1976.

[103] A. Van Gelder. The alternating fixpoint of logic programs with negation: Extended abstract. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1989.

[104] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *J. Logic Program.*, 6:109–133, 1989.

[105] D. E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22:269–301, 1984.

[106] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, San Mateo, CA, 1988.

[107] R. Zabih. Some applications of graph bandwidth to constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 46–51, 1990.

[108] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 155–160, 1988.

DISTRIBUTION LIST

addresses	number of copies
DR NORTHRUP FOWLER III ROME LABORATORY/C3C 525 BROOKS ROAD GRIFFISS AFB NY 1341-4505	10
STANFORD UNIVERSITY ATTN: DR NILS NILSSON COMPUTER SCIENCE DEPARTMENT STANFORD CA 94305-2140	5
RL/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY GRIFFISS AFB NY 13441-4514	1
ADMINISTRATOR DEFENSE TECHNICAL INFO CENTER DTIC-FDAC CAMERON STATION BUILDING 5 ALEXANDRIA VA 22304-6145	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
RL/C3AB 525 BROOKS RD GRIFFISS AFB NY 13441-4505	1
NAVAL WARFARE ASSESSMENT CENTER GIDEP OPERATIONS CENTER/CODE QA-50 ATTN: E RICHARDS CORONA CA 91718-5000	1
ASC/ENEMS WRIGHT-PATTERSON AFB OH 45433-6503	1

WRIGHT LABORATORY/AAAI-2 1
ATTN: MR FRANKLIN HUTSON
WRIGHT-PATTERSON AFB OH 45433-6543

AFIT/LDEE 1
2950 P STREET
WRIGHT-PATTERSON AFB OH 45433-6577

WRIGHT LABORATORY/MTEL 1
WRIGHT-PATTERSON AFB OH 45433

AAMRL/HE 1
WRIGHT-PATTERSON AFB OH 45433-6573

AIR FORCE HUMAN RESOURCES LAB 1
TECHNICAL DOCUMENTS CENTER
AFHRL/LRS-TDC
WRIGHT-PATTERSON AFB OH 45433

AUL/LSE 1
BLDG 1405
MAXWELL AFB AL 36112-5564

US ARMY STRATEGIC DEF 1
CSSD-IM-PA
PO BOX 1500
HUNTSVILLE AL 35807-3801

COMMANDING OFFICER 1
NAVAL AVIONICS CENTER
LIBRARY D/765
INDIANAPOLIS IN 46219-2189

COMMANDING OFFICER 1
NCCOSC RDTE DIVISION
CODE 0274B, TECH LIBRARY
53560 HULL STREET
SAN DIEGO CA 92152-5001

CMDR 1
NAVAL WEAPONS CENTER
TECHNICAL LIBRARY/C3431
CHINA LAKE CA 93555-6001

SPACE & NAVAL WARFARE SYSTEMS COMM 1
WASHINGTON DC 20363-5100

CDR, U.S. ARMY MISSILE COMMAND 2
REDSTONE SCIENTIFIC INFO CENTER
AMSMI-RD-CS-R/ILL DOCUMENTS
REDSTONE ARSENAL AL 35898-5241

ADVISORY GROUP ON ELECTRON DEVICES 2
ATTN: DOCUMENTS
2011 CRYSTAL DRIVE,SUITE 307
ARLINGTON VA 22202

REPORT COLLECTION, RESEARCH LIBRARY 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

AEDC LIBRARY 1
TECH FILES/MS-100
ARNOLD AFB TN 37389

COMMANDER/USAISC 1
ATTN: ASOP-DO-TL
BLDG 61801
FT HUACHUCA AZ 85613-5000

AIR WEATHER SERVICE TECHNICAL LIB 1
FL 4414
SCOTT AFB IL 62225-5458

AFIWC/MSO 1
102 HALL BLVD STE 315
SAN ANTONIO TX 78243-7016

SOFTWARE ENGINEERING INST (SEI) 1
TECHNICAL LIBRARY
5000 FORBES AVE
PITTSBURGH PA 15213

DIRECTOR NSA/CSS 1
W157
9800 SAVAGE ROAD
FORT MEADE MD 21055-6000

NSA 1
ATTN: D. ALLEY
DIV X911
9800 SAVAGE ROAD
FT MEADE MD 20755-6000

DOD 1
R31
9800 SAVAGE ROAD
FT. MEADE MD 20755-6000

DIRNSA 1
R509
9800 SAVAGE ROAD
FT MEADE MD 20775

DOD COMPUTER CENTER 1
C/TIC
9800 SAVAGE ROAD
FORT GEORGE G. MEADE MD 20755-6000

ESC/IC 1
50 GRIFFISS STREET
HANSCOM AFB MA 01731-1619

DCMAO/GWE 1
ATTN: JOHN CHENG
US COURTHOUSE/SUITE B-34
401 N MARKET
WICHITA KS 67202-2095

FL 2807/RESEARCH LIBRARY 1
OL AA/SULL
HANSCOM AFB MA 01731-5000

TECHNICAL REPORTS CENTER 1
MAIL DROP D130
BURLINGTON ROAD
BEDFORD MA 01731

DEFENSE TECHNOLOGY SEC ADMIN (DTSA) 1
ATTN: STTD/PATRICK SULLIVAN
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

SOFTWARE ENGR'G INST TECH LIBRARY 1
ATTN: MR DENNIS SMITH
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213-3890

SOFTWARE OPTIONS, INC. 1
ATTN: MR TOM CHEATHAM
22 HILLIARD STREET
CAMBRIDGE MA 02138

USC-IST 1
ATTN: DR ROBERT M. BALZER
4676 ADMIRALTY WAY
MARINA DEL REY CA 90292-6695

KESTREL INSTITUTE 1
ATTN: DR CORDELL GREEN
1801 PAGE MILL ROAD
PALO ALTO CA 94304

ROCHESTER INSTITUTE OF TECHNOLOGY 1
ATTN: PROF J. A. LASKY
1 LOMB MEMORIAL DRIVE
P.O. BOX 9887
ROCHESTER NY 14613-5700

WESTINGHOUSE ELECTRONICS CORP 1
ATTN: MR DENNIS BIELAK
ELECTRONICS SYSTEMS GROUP
P.O. BOX 746, MAIL STOP 432
BALTIMORE MD 21203

AFIT/ENG 1
ATTN: PAUL BAILOR. MAJOR, USAF
WPAFB OH 45433-6583

THE MITRE CORPORATION 1
ATTN: MR EDWARD H. BENSLEY
BURLINGTON RD/MAIL STOP A350
BEDFORD MA 01730

UNIV OF ILLINOIS, URBANA-CHAMPAIGN 1
ATTN: SANJAY BHANSALI
DEPT OF COMPUTER SCIENCES
1304 WEST SPRINGFIELD
URBANA IL 61801

ANDERSEN CONSULTING 1
ATTN: MR MICHAEL E. DEBELLIS
100 SOUTH WACKER DRIVE
CHICAGO IL 60606

UNIV OF ILLINOIS, URBANA-CHAMPAIGN 1
ATTN: DR MEHDI HARANDI
DEPT OF COMPUTER SCIENCES
1304 W. SPRINGFIELD/240 DIGITAL LAB
URBANA IL 61801

HONEYWELL, INC. 1
ATTN: MR BERT HARRIS
FEDERAL SYSTEMS
7900 WESTPARK DRIVE
MCLEAN VA 22102

SOFTWARE ENGINEERING INSTITUTE 1
ATTN: MR WILLIAM E. HEFLEY
CARNEGIE-MELLON UNIVERSITY
SEI 2218
PITTSBURGH PA 15213-38990

UNIVERSITY OF SOUTHERN CALIFORNIA 1
ATTN: DR W. LEWIS JOHNSON
INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY/SUITE 1001
MARINA DEL REY CA 90292-6695

COLUMBIA UNIV/DEPT COMPUTER SCIENCE 1
ATTN: DR GAIL E. KAISER
450 COMPUTER SCIENCE BLDG
500 WEST 120TH STREET
NEW YORK NY 10027

SOFTWARE ENGINEERING INSTITUTE 1
ATTN: KYO CHUL KANG
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH PA 15213-3890

SOFTWARE PRODUCTIVITY CONSORTIUM 1
ATTN: MR ROBERT LAI
2214 ROCK HILL ROAD
HERNDON VA 22070

AFIT/ENG 1
ATTN: DR GARY B. LAMONT
SCHOOL OF ENGINEERING
DEPT ELECTRICAL & COMPUTER ENGRG
WPAFB OH 45433-6583

NSA/OFC OF RESEARCH 1
ATTN: MS MARY ANNE OVERMAN
9800 SAVAGE ROAD
FT GEORGE G. MEADE MD 20755-6000

THE MITRE CORPORATION 1
ATTN: MR HOWARD REUBENSTEIN
BURLINGTON ROAD
BEDFORD MA 01730

AT&T BELL LABORATORIES 1
ATTN: MR PETER G. SELFRIDGE
ROOM 3C-441
600 MOUNTAIN AVE
MURRAY HILL NJ 07974

VITRO CORPORATION 1
ATTN: MR ROBERT A. SMALL
14000 GEORGIA AVENUE
SILVER SPRING MD 20906-2972

ODYSSEY RESEARCH ASSOCIATES, INC. 1
ATTN: MS MAUREEN STILLMAN
301A HARRIS B. DATES DRIVE
ITHACA NY 14850-1313

WRDC/AAAF-3 1
ATTN: JAMES P. WEBER, CAPT, USAF
AERONAUTICAL SYSTEMS CENTER
WPAFB OH 45433-6543

TEXAS INSTRUMENTS INCORPORATED 1
ATTN: DR DAVID L. WELLS
P.O. BOX 655474, MS 238
DALLAS TX 75265

BOEING COMPUTER SERVICES 1
ATTN: DR PHIL NEWCOMB
MS 7L-64
P.O. BOX 24346
SEATTLE WA 98L24-0346

LOCKHEED SOFTWARE TEHNOLOGY CENTER 1
ATTN: MR HENSON GRAVES
ORG. 96-L0 BLDG 254E
3251 HANOVER STREET
PALO ALTO CA 94304-LL9L

REASONING SYSTEMS 1
ATTN: DR GORDON KOTIK
3260 HILLVIEW AVENUE
PALO ALTO CA 94304

TEXAS A & M UNIVERSITY 1
ATTN: DR PAULA MAYER
KNOWLEDGE BASED SYSTEMS LABORATORY
DEPT OF INDUSTRIAL ENGINEERING
COLLEGE STATION TX 77843

KESTREL DEVELOPMENT CORPORATION 1
ATTN: DR RICHARD JULLIG
3260 HILLVIEW AVENUE
PALO ALTO CA 94304

DARPA/TTO 1
ATTN: DV
1400 WILSON BLVD
ARLINGTON VA 22203-2309

ARPA/SISTO 1
ATTN: DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714

LOCKHEED O/96-10 B/254E 1
ATTN: JACKY COMBS
3251 HANOVER STREET
PALO ALTO CA 94304-1191

NASA/JOHNSON SPACE CENTER 1
ATTN: CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058

SAIC 1
ATTN: LANCE MILLER
MS T1-6-3
PO BOX 1303 (OR 1710 GOODRIDGE DR)
MCLEAN VA 22102

STERLING IMD INC. 1
KSC OPERATIONS
ATTN: MARK MAGINN
BEECHES TECHNICAL CAMPUS/RT 26 N.
ROME NY 13440

NAVAL POSTGRADUATE SCHOOL 1
ATTN: BALA RAMESH
CODE AS/RS
ADMINISTRATIVE SCIENCES DEPT
MONTEREY CA 93943

KESTREL INSTITUTE 1
ATTN: MARIA PRYCE
3260 HILLVIEW AVENUE
PALO ALTO CA 94304

HUGHES AIRCRAFT COMPANY 1
ATTN: GERRY BARKSDALE
P. O. BOX 3310
BLDG 618 MS E215
FULLERTON CA 92634

FORWISS UNIVERSITY OF ERLANGEN 1
ATTN: ERNST LUTZ
AM WEICHSELGARTEN 7
8520 ERLANGEN, GERMANY

THE MITRE CORPORATION 1
ATTN: HOWARD REUBENSTEIN
BURLINGTON ROAD, K302
BEDFORD MA 01730

SCHLUMBERGER LABORATORY FOR 1
COMPUTER SCIENCE
ATTN: DR. GUILLERMO ARANGO
8311 NORTH FM620
AUSTIN, TX 78720

PARAMAX SYSTEMS CORPORATION 1
ATTN: DON YU
8201 GREENSBORO DRIVE, SUITE 1000
MCLEAN VA 22101

MOTOROLA, INC. 1
ATTN: MR. ARNOLD PITTLER
3701 ALGONQUIN ROAD, SUTE 601
ROLLING MEADOWS, IL 60008

DECISION SYSTEMS DEPARTMENT 1
ATTN: PROF WALT SCACCHI
SCHOOL OF BUSINESS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90089-1421

SOUTHWEST RESEARCH INSTITUTE 1
ATTN: BRUCE REYNOLDS
6220 CULEBRA ROAD
SAN ANTONIO, TX 78228-0510

NATIONAL INSTITUTE OF STANDARDS 1
AND TECHNOLOGY
ATTN: CHRIS DABROWSKI
ROOM A266, BLDG 225
GAITHSBURG MD 20899

EXPERT SYSTEMS LABORATORY 1
ATTN: STEVEN H. SCHWARTZ
NYNEX SCIENCE & TECHNOLOGY
500 WESTCHESTER AVENUE
WHITE PLANS NY 20604

NAVAL TRAINING SYSTEMS CENTER 1
ATTN: ROBERT BREAUX/CODE 252
12350 RESEARCH PARKWAY
ORLANDO FL 32826-3224

CENTER FOR EXCELLENCE IN COMPUTER- 1
AIDED SYSTEMS ENGINEERING
ATTN: PERRY ALEXANDER
2291 IRVING HILL ROAD
LAWRENCE KS 66049

SOFTWARE TECHNOLOGY SUPPORT CENTER 1
ATTN: MAJ ALAN K. MILLER
OGDEN ALC/TISE
BLDG 100, BAY G
HILL AFB, UTAH 84056

MS. KAREN ALGUIRE 1
RL/C3CA
525 BROOKS RD
GRIFFISS AFB NY 13441-4505

JAMES ALLEN 1
COMPUTER SCIENCE DEPT/BLDG RM 732
UNIV OF ROCHESTER
WILSON BLVD
ROCHESTER NY 14627

YIGAL ARENS 1
USC-ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

MR. RAY BAREISS 1
THE INST. FOR LEARNING SCIENCES
NORTHWESTERN UNIV
1890 MAPLE AVE
EVANSTON IL 60201

MR. JEFF BERLINER 1
BBN SYSTEMS & TECHNOLOGIES
10 MOULTON STREET
CAMBRIDGE MA 02138

MARIE A. BIENKOWSKI 1
SRI INTERNATIONAL
333 RAVENSWOOD AVE/EK 337
MENLO PRK CA 94025

DR MARK S. BODDY 1
HONEYWELL SYSTEMS & RSCH CENTER
3660 TECHNOLOGY DRIVE
MINNEAPOLIS MN 55418

PIERO P. BONISSONE 1
GE CORPORATE RESEARCH & DEVELOPMENT
BLDG K1-RM 5C-32A
P. O. BOX 8
SCHENECTADY NY 12301

MR. DAVID BROWN 1
MITRE
EAGLE CENTER 3, SUITE 8
O'FALLON IL 62269

MR. MARK BURSTEIN 1
BBN SYSTEMS & TECHNOLOGIES
10 MOULTON STREET
CAMBRIDGE MA 02138

MR. GREGG COLLINS 1
INST FOR LEARNING SCIENCES
1890 MAPLE AVE
EVANSTON IL 60201

MR. RANDALL J. CALISTRI-YEH 1
ORA CORPORATION
301 DATES DRIVE
ITHACA NY 14850-1313

DR STEPHEN E. CROSS 1
SCHOOL OF COMPUTER SCIENCE
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213

MS. JUDITH DALY 1
ARPA/ASTO
3701 N. FAIRFAX DR., 7TH FLOOR
ARLINGTON VA 22203-1714

THOMAS CHEATHAM 1
HARVARD UNIVERSITY
DIV OF APPLIED SCIENCE
AIKEN, RM 104
CAMBRIDGE MA 02138

MS. LAURA DAVIS 1
CODE 5510
NAVY CTR FOR APPLIED RES IN AI
NAVAL RESEARCH LABORATORY
WASH DC 20375-5337

MS. GLADYS CHOW 1
COMPUTER SCIENCE DEPT.
UNIV OF CALIFORNIA
LOS ANGELES CA 90024

THOMAS L. DEAN 1
BROWN UNIVERSITY
DEPT OF COMPUTER SCIENCE
P.O. BOX 1910
PROVIDENCE RI 02912

WESLEY CHU 1
COMPUTER SCIENCE DEPT
UNIV OF CALIFORNIA
LOS ANGELES CA 90024

MR. ROBERTO DESIMONE 1
SRI INTERNATIONAL (EK335)
333 RAVENSWOOD AVE
MENLO PRK CA 94025

PAUL R. COHEN 1
UNIV OF MASSACHUSETTS
COINS DEPT
LEDERLE GRC
AMHERST MA 01003

MS. MARIE DEJARDINS 1
SRI INTERNATIONAL
333 RAVENSWOOD AVENUE
MENLO PRK CA 94025

JON DOYLE 1
LABORATORY FOR COMPUTER SCIENCE
MASS INSTITUTE OF TECHNOLOGY
545 TECHNOLOGY SQUARE
CAMBRIDGE MA 02139

DR. BRIAN DRABBLE 1
AI APPLICATIONS INSTITUTE
UNIV OF EDINBURGH/80 S. BRIDGE
EDINBURGH EH1 LHN
UNITED KINGDOM

MR. SCOTT FOUSE 1
TSX CORPORATION
4353 PARK TERRACE DRIVE
WESTLAKE VILLAGE CA 91361

MR. STU DRAPER 1
MITRE
EAGLE CENTER 3, SUITE 8
O'FALLON IL 62269

MARK FOX 1
DEPT O INDUSTRIAL ENGRG
UNIV OF TORONTO
4 TADDLE CREAK ROAD
TORONTO, ONTARIO, CANADA

MR. GARY EDWARDS 1
4353 PARK TERRACE DRIVE
WESTLAKE VILLACA 91361

MS. MARTHA FARINACCI 1
MITRE
7525 COLSHIRE DRIVE
MCLEAN VA 22101

MR. RUSS FREW 1
GENERAL ELECTRIC
MOORESTOWN CORPORATE CENTER
BLDG ATK 145-2
MOORESTOWN NJ 08057

MICHAEL FEHLING 1
STANFORD UNIVERSITY
ENGINEERING ECO SYSTEMS
STANFORD CA 94305

MR. RICH FRITZSON 1
CENTER OR ADVANCED INFO TECHNOLOGY
UNISYS
P.O. BOX 517
PAOLI PA 19301

MR KRISTIAN J. HAMMOND 1
UNIV OF CHICAGO
COMPUTER SCIENCE DEPT/RY155
1100 E. 58TH STREET
CHICAGO IL 60637

MR. ROBERT FROST 1
MITRE CORP
WASHINGTON C3 CENTER, MS 644
7525 COLSHIER ROAD
MCLEAN VA 22101-3481

RICK HAYES-ROTH 1
CIMFLEX-TEKNOWLEDGE
1810 EMBARCADERO RD
PALO ALTO CA 94303

RANDY GARRETT 1
INST FOR DEFENSE ANALYSES (IDA)
1801 N. BEAUREGARD STREET
ALEXANDRA VA 22311-1772

MR. JIM HENDLER 1
UNIV OF MARYLAND
DEPT OF COMPUTER SCIENCE
COLLEGE PARK MD 20742

MS. YOLANDA GIL 1
USC/ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

MR.MAX HERION 1
ROCKWELL INTERNATIONAL SCIENCE CTR
444 HIGH STREET
PALO ALTO CA 94301

MR. STEVE GOYA 1
DISA/JIEO/GS11
CODE TBD
11440 ISAAC NEWTON SQ
RESTON VA 22090

MR. MORTON A. HIRSCHBERG, DIRECTOR 1
US ARMY RESEARCH LABORATORY
ATTN; AMSRL-CI-CB
ABERDEEN PROVING GROUND MD
21005-5066

MR. MARK A. HOFFMAN 1
ISX CORPORATION
1165 NORTHCHASE PARKWAY
MARIETTA GA 30067

MR. RON LARSEN 1
NAVAL CMD, CONTROL & OCEAN SUR CTR
RESEARCH, DEVELOP, TEST & EVAL DIV
CODE 444
SAN DIEGO CA 92152-5000

DR. JAMES JUST 1
MITRE
DEPT. W032-M/S Z360
7525 COLSHIER RD
MCLEAN VA 22101

MR. CRAIG KNOBLOCK 1
USC-ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

MR. RICHARD LOWE (AP-10) 1
SRA CORPORATION
2000 15TH STREET NORTH
ARLINGTON VA 22201

MR. TED C. KRAL 1
BBN SYSTEMS & TECHNOLOGIES
4015 HANCOCK STREET, SUITEE 101
SAN DIEGO CA 92110

MR. JOHN LOWRENCE 1
SRI INTERNATIONAL
ARTIFICIAL INTELLIGENCE CENTER
333 RAVENSWOOD AVE
MENLO PARK CA 94025

DR. ALAN MEYROWITZ 1
NAVAL RESEARCH LABORATORY/CODE 5510
4555 OVERLOOK AVE
WASH DC 20375

ALICE MULVEHILL 1
MITRE CORPORATION
BURLINGTON RD
M/S K-302
BEDFORD MA 01730

ROBERT MACGREGOR 1
USC/ISI
4676 ADMIRALTY WAY
MARINA DEL REY CA 90292

WILLIAM S. MARK, MGR AI CENTER 1
LOCKHEED MISSILES & SPACE CENTER
1801 PAGE MILL RD
PALO ALTO CA 94304-1211

RICHARD MARTIN 1
SOTWARE ENGINEERING INSTITUTE
CARNEGIE MELLON UNIV
PITTSBURGH PA 16213

DREW MCDERMOTT 1
YALE COMPUTER SCIENCE DEPT
P.O. BOX 2158, YALE STATION
51 PROPSPECT STREET
MEW HAVEN CT 06520

MS. CECILE PARIS 1
USC/ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

DOUGLAS SMITH 1
KESTREL INSTITUTE
3260 HILLVIEW AVE
PALO ALTO CA 94304

DR. AUSTIN TATE 1
AI APPLICATIONS INSTITUTE
UNIV OF EDINBURGH
80 SOUTH BRIDGE
EDINBURGH EH1 1HN - SCOTLAND

EDWARD THOMPSON 1
ARPA/SISTO
3701 N. FAIRFAX DR., 7TH FL
ARLINGTON VA 22209-1714

MR. STEPHEN F. SMITH 1
ROBOTICS INSTITUTE/CMU
SCHENLEY PRK
PITTSBURGH PA 15213

DR. ABRAHAM WAKSMAN 1
AFOSR/NM
110 DUNCAN AVE., SUITE B115
BOLLING AFB DC 20331-0001

JONATHAN P.STILLMAN 1
GENERAL ELECTRIC CRD
1 RIVER RD, RM K1-5C31A
P. O. BOX 8
SCHENECTADY NY 12345

MR. EDWARD C. T. WALKER 1
BBN SYSTEMS & TECHNOLOGIES
10 MOULTON STREET
CAMBRIDGE MA 02138

MR. BILL SWARTOUT 1
USC/ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

GIO WIEDERHOLD 1
STANFORD UNIVERSITY
DEPT OF COMPUTER SCIENCE
438 MARGARET JACKS HALL
STANFORD CA 94305-2140

KATIA SYCARA/THE ROBOTICS INST 1
SCHOOL OF COMPUTER SCIENCE
CARNEGIE MELLON UNIV
DOHERTY HALL RM 3325
PITTSBURGH PA 15213

MR. DAVID E. WILKINS 1
SRI INTERNATIONAL
ARTIFICIAL INTELLIGENCE CENTER
333 RAVENSWOOD AVE
MENLO PARK CA 94025

DR. PATRICK WINSTON 1
MASS INSTITUTE OF TECHNOLOGY
RM NE43-817
545 TECHNOLOGY SQUARE
CAMBRIDGE MA 02139

HUA YANG 1
COMPUTER SCIENCE DEPT
UNIV OF CALIORNIA
LOS ANGELES CA 90024

LTCOL DAVE NEYLAND 1
ARPA/ISTO
3701 N. FAIRFAX DRIVE, 7TH FLOOR
ARLINGTON VA 22209-1714

MR. RICK SCHANTZ 1
BBN SYSTEMS & TECHNOLOGIES
10 MOULTON STREET
CAMBRIDGE MA 02138

LTC FRED M. RAWCLIFFE 1
USTRANSCOM/TCJ5-SC
BLDG 1900
SCOTT AFB IL 62225-7001

ARPA/SISTO 1
ATTN: MR JOHN P. SCHILL
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714

MR. DONALD F. ROBERTS 1
RL/C3CA
525 BROOKS ROAD
GRIFFISS AFB NY 13441-4505

ALLEN SEARS 1
MITRE
7525 COLESHIRE DRIVE, STOP Z289
MCLEAN VA 22101

STEVE ROTH 1
CENTER FOR INTEGRATED MANUFACTURING
THE ROBOTICS INSTITUTE
CARNEGIE MELLON UNIV
PITTSBURGH PA 15213-3890

JEFF ROTHENBERG 1
SENIOR COMPUTER SCIENTIST
THE RAND CORPORATION
1700 MAIN STREET
SANTA MONICA CA 90407-2138

YOAV SHOHAM 1
STANFORD UNIVERSITY
COMPUTER SCIENCE DEPT
STANFORD CA 94305

MR. DAVID B. SKALAK 1
UNIV OF MASSACHUSETTS
DEPT OF COMPUTER SCIENCE
RM 243, LGRC
AMHERST MA 01003

MR. MIKE ROUSE 1
AFSC
7800 HAMPTON RD
NORFOLK VA 23511-6097

MR. DAVID E. SMITH 1
ROCKWELL INTERNATIONAL
444 HIGH STREET
PALO ALTO CA 94301

JEFF ROTHENBERG 1
SENIOR COMPUTER SCIENTIST
THE RAND CORPORATION
1700 MIN STREET
SANTA MONICA CA 90407-2138

DR LARRY BIRNBAUM 1
NORTHWESTERN UNIVERSITY
ILS
1890 MAPLE AVE
EVANSTON IL 60201

MR RANDALL J. CALISTRI-YEH 1
ORA
301 DATES DR
ITHACA NY 14850-1313

MR WESLEY CHU 1
COMPUTER SCIENCE DEPT
UNIVERSITY OF CALIFORNIA
LOS ANGELES CA 9002

MR PAUL R COHEN 1
UNIVERSITY OF MASSACHUSETTS
COINS DEPT, LEDERLE GRC
AMHERST MA 01003

MR DON EDDINGTON 1
NAVAL COMMAND, CONTROL & OCEAN
SURV CENTER
RDT&E DIVISION, CODE 404
SAN DIEGO CA 92152-5000

MR. LEE ERMAN 1
CIMFLEX TECKNOWLEDGE
1810 EMBARCARDERO RD
PALO ALTO CA 94303

MR DICK ESTRADA 1
BBN SYSTEMS & TECHNOLOGIES
10 MOULTON ST
CAMBRIDGE MA 02138

MR HARRY FORSDICK 1
BBN SYSTEMS AND TECHNOLOGIES
10 MOULTON ST
CAMBRIDGE MA 02138

MR MATTHEW L. GINSBERG 5
CIRL, 1269
UNIVERSITY OF OREGON
EUGENE OR 97403

MR IRA GOLDSTEIN 1
OPEN SW FOUNDATION RESEARCH INST
ONE CAMBRIDGE CENTER
CAMBRIDGE MA 02142

MR MOISES GOLDSZMIDT 1
INFORMATION AND DECISION SCIENCES
ROCKWELL INTL SCIENCE CENTER
444 HIGH ST, SUITE 400
PALO ALTO CA 94301

MR JEFF GROSSMAN, CO 1
NCCOSC RDTE DIV 44
5370 SILVERGATE AVE, ROOM 1405
SAN DIEGO CA 92152-5146

JAN GUNTHER 1
ASCENT TECHNOLOGY, INC.
64 SIDNEY ST, SUITE 380
CAMBRIDGE MA 02139

DR LYNETTE HIRSCHMAN 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

MS ADELE E. HOWE 1
COMPUTER SCIENCE DEPT
COLORADO STATE UNIVERSITY
FORT COLLINS CO 80523

DR LESLIE PACK KAELBLING 1
COMPUTER SCIENCE DEPT
BROWN UNIVERSITY
PROVIDENCE RI 02912

SUBBARAO KAMBHAMPATI 1
DEPT OF COMPUTER SCIENCE
ARIZONA STATE UNIVERSITY
TEMPE AZ 85287-5406

MR THOMAS E. KAZMIERCZAK 1
SRA CORPORATION
331 SALEM PLACE, SUITE 200
FAIRVIEW HEIGHTS IL 62208

PRADEEP K. KHOSLA 1
ARPA/SSTO
3701 N. FAIRFAX DR
ARLINGTON VA 22203

MR CRAIG KNOBLOCK 1
USC-ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

DR CARLA LUDLOW 1
ROME LABORATORY/C3CA
525 BROOKS RD
GRIFFISS AFB NY 13441-4505

DR MARK T. MAYBURY 1
ASSOCIATE DIRECTOR OF AI CENTER
ADVANCED INFO SYSTEMS TECH G041
MITRE CORP, BURLINGTON RD, MS K-329
BEDFORD MA 01730

MR DONALD P. MCKAY 1
PARAMAX/UNISYS
P O BOX 517
PAOLI PA 19301

DR KAREN MYERS 1
AI CENTER
SRI INTERNTIONAL
333 RAVENSWOOD
MENLO PARK CA 94025

DR MARTHA E POLLACK 1
DEPT OF COMPUTER SCIENCE
UNIVERSITY OF PITTSBURGH
PITTSBURGH PA 15260

RAJ REDDY 1
SCHOOL OF COMPUTER SCIENCE
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213

EDWINA RISSLAND 1
DEPT OF COMPUTER & INFO SCIENCE
UNIVERSITY OF MASSACHUSETTS
AMHERST MA 01003

MR NORMAN SADEH 1
CIMDS
THE ROBOTICS INSTITUTE
CARNEGTE MELLON UNIVERSITY
PITTSBURGH PA 15213

MR ERIC TIFFANY 1
ASCENT TECHNOLOGY INC.
237 LONGVIEW TERRACE
WILLIAMSTOWN MA 01267

MANUELA VELOSO 1
CARNEGIE MELLON UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
PITTSBURGH PA 15213-3891

MR DAN WELD 1
DEPT OF COMPUTER SCIENCE & ENG
MAIL STOP FR-35
UNIVERSITY OF WASHINGTON
SEATTLE WA 98195

MR CRAIG WIER 1
ARPA/SISTO
3701 N. FAIRFAX DR
ARLINGTON VA 22203

MR JOE ROBERTS 1
ISX CORPORATION
4301 N FAIRFAX DRIVE, SUITE 301
ARLINGTON VA 22203

COL JOHN A. WARDEN III 1
ASC/CC
225 CHENNAULT CIRCLE
MAXWELL AFB AL 36112-6426

DR TOM GARVEY 1
ARPA/SISTO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

MR JOHN N. ENTZMINGER, JR. 1
ARPA/DIRO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

LT COL ANTHONY WAISANEN, PHD 1
COMMAND ANALYSIS GROUP
HQ AIR MOBILITY COMMAND
402 SCOTT DRIVE, UNIT 3L3
SCOTT AFB IL 62225-5307

DIRECTOR 1
ARPA/SISTO
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

MS LESLIE WILLIAMS 1
DIGITAL SYSEMS RSCH INC
4301 NORTH FAIRFAX DRIVE
SUITE 725
ARLINGTON VA 22203

DECISIONS & DESIGNS INC. 1
ATTN: ANN MARTIN
8219 LEESBURG PIKE
SUITE 390
VIENNA VA 22182

MS LEAH WONG 5
NCCOSC RDTE DIV
53570 SILVERGATE AVE
SAN DIEGO CA 92152-5246

☆U.S. GOVERNMENT PRINTING OFFICE: 1995-610-126-50174

Rome Laboratory

Customer Satisfaction Survey

RL-TR-________

Please complete this survey, and mail to RL/IMPS,
26 Electronic Pky, Griffiss AFB NY 13441-4514. Your assessment and feedback regarding this technical report will allow Rome Laboratory to have a vehicle to continuously improve our methods of research, publication, and customer satisfaction. Your assistance is greatly appreciated.
Thank You

____

____

Organization Name:_____________________________________(Optional)

Organization POC: _____________________________________(Optional)

Address:____

1. On a scale of 1 to 5 how would you rate the technology developed under this research?

5-Extremely Useful   1-Not Useful/Wasteful

Rating_____

Please use the space below to comment on your rating. Please suggest improvements. Use the back of this sheet if necessary.

2. Do any specific areas of the report stand out as exceptional?

Yes___ No______

If yes, please identify the area(s), and comment on what aspects make them "stand out."

3. Do any specific areas of the report stand out as inferior?

Yes___ No___

If yes, please identify the area(s), and comment on what aspects make them "stand out."

4. Please utilize the space below to comment on any other aspects of the report. Comments on both technical content and reporting format are desired.

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.